Model checking of component connectors

Izadi, M.

Model Checking of Component Connectors

Mohammad Izadi

Model Checking of Component Connectors

proefschrift

ter verkrijging van

de graad van Doctor aan de Universiteit Leiden,

op gezag van de Rector Magnificus prof. mr. P.F. van der Heijden,

volgens besluit van het College voor Promoties

te verdedigen op dinsdag 6 december 2011

klokke 13:45 uur

door

Mohammad Izadi

geboren te Najafabad (Iran)

in 1972

Promotiecommissie

Promotors:	Prof. Dr. Farhad Arbab
	Prof. Dr. Ali Movaghar (*Sharif University of Technology, Iran*)
Co-promotor:	Dr. Marcello M. Bonsangue
Overige leden:	Prof. Dr. Joost N. Kok
	Prof. Dr. Frank de Boer
	Dr. Natallia Kokash
	Dr. Marjan Sirjani (*Reykjavik University, Iceland*)

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA Dissertation Series 2011-22

To *my faithful lovely wife,*
Faeze
and ***for*** *our beloved children,*
our daughter, ***Negar***
and our little son, ***Nikan***

Contents

List of Figures

List of Tables

1 Introduction

The concept of *component based systems*, especially component-based software, is a new philosophy or way of thinking to deal with the complexity in designing large scale computing systems[13, 135]. One of the main goals of this approach is to compose reusable components by some glue codes. The model according to which these *components* are composed is called *coordination model*. *Coordination languages* are specification languages for coordination models. Reo is a coordination language which is based on a calculus of channels [13]. By using Reo specifications, complex component connectors can be organized as a network of channels and built in a compositional manner.

In this thesis, we investigate the formal verification of properties of Reo coordination models. The main question we address is: how can the desired properties of a coordination model specified in Reo be *formally* verified. In particular, the problem is interesting if the state space of the model is very large. To answer this question, we investigate an automata-theoretic model checking method for Reo specifications in the presence of the state explosion problem. In this way we first propose a formal semantics of Reo based on a generalization of the standard notion of Büchi automata and containing some ideas from constraint automaton (the first operational model for Reo [30]). In the following section, we introduce the context of this research and the main questions more precisely. Also, in the forthcoming sections, we introduce our main thesis and its motivations and a history of the research in this field. In the last section of this chapter, we introduce the outline of the thesis and for each chapter its contributions.

1.1 Research Context and Main Question

The work of this thesis is categorized under the computer science fields of *formal verification* and *coordination systems*. In fact, this thesis introduces a formal verification framework for coordination systems in particular and for *component-based systems* in general. Let us introduce these fields briefly.

A system that consists of a set of independent computing components and a coordinating subsystem is called a component based system. Coordination is defined as the study of the dynamic topologies of interactions among concurrent programs, processes and components of a system, with the goal of finding solutions to the problem of managing these interactions [12]. To be more precise about the coordination systems, we need to model or express the coordination strategies using some kind of modeling formalism or language. There exist many coordination models and languages in the literature [120]. In this thesis we concentrate on the coordination language *Reo*, that is, an exogenous coordination language which can specify coordination of a set of components through networks of channels or compositional connectors built out of primitive ones [13].

The main goal of verification methods is trying to ascertain that an actual system or program satisfies its requirements. In *formal* verification one tries to achieve the aim of verification by describing the system using a mathematical model, expressing the requirements as properties of this model and by showing through rigorous mathematical reasoning that the model of the system indeed has the required properties [50, 110]. There are two main methods for formal verification: *deductive verification* (theorem proving) and *model checking*. In de-

ductive verification, the system to be proved correct is described by a set of formulas called *axioms*. The process of verification is to derive a proof of the desired correctness property based on the set of axioms. In model checking, the system to be proved correct is modeled by a kind of finite transition system and the desired property is expressed by a formula in a formal language such as the language of a temporal logic. The process of verification is to check all of the state space of the model for the satisfaction of the property's formula. In this thesis we will consider verification methods that fall into the field of model checking, because they can be fully automated and hence more suitable for an implementation.

Thus, our main question or goal is *to find a model checking verification framework for coordination systems specified by Reo*. For this aim, the coordination systems specified by Reo should be modeled by some sort of an operational (transition systems-based) model. There are a number of operational semantic models for Reo, such as constraint automata [30], intentional automata [52] and Reo automata [36, 37]. However, these models have shortcomings in fully expressing some aspects of coordinations such as synchronization of I/O operations, context dependency, and fairness constraints or are not very suitable for model checking. Therefore, we present a new operational semantics for Reo called *Büchi automata of records (BAR)* and its augmented versions.

When the modeling formalism is a sort of automata on infinite objects (such as Büchi automata on infinite strings which is our selected formalism), the most suitable model checking method is that of *automata theoretic model checking*. In this method, the negation of the desired property is expressed by an automaton (directly or after translation from formulas of a temporal logic such as LTL) and the emptiness of the language of the (intersection) product of the two automata (system and property automata) is checked.

The model checking process can suffer from the problem of *state explosion* since the model of the system tends to be extremely large. We select two independent methods to tackle this problem. The first is to implement the state space symbolically using ordered binary decision diagrams (OBDD) and running the model checking algorithm over them, a method called *symbolic model checking*. The other is to minimize the models based on some proper equivalence relations, called *compositional minimization*. Also, when we obtain the property automaton from translation of linear temporal logic formulas, the translation can be done not only inductively but also by using an *on-the-fly* method.

1.2 This Thesis

In this thesis, we present a framework for automata theoretic model checking of coordination systems specified in Reo. As an operational modeling formalism that covers several intended behaviors of Reo connectors such as fairness, I/O synchronization, and context dependency, we introduce Büchi automata of records (BAR) and their augmented version (ABAR). We show that every constraint automaton (the first introduced operational semantics of Reo) can be translated into an essentially equivalent BAR. However, there are some Reo connectors' behaviors expressible by BAR's that constraint automata are not able to express.

To specify the properties to be verified, we introduce an action based linear temporal

logic called ρ-LTL interpreted over the executions of augmented Büchi automata of records and show how its formulas can be translated into their equivalent ABAR's. The translation can be done inductively or by using an on-the-fly method. To deal with the large state spaces, we show that ABAR's can be implemented using ordered binary decision diagrams (OBDD) as their dense data structures. For this purpose, we also introduce the necessary modifications over the basic model checking algorithm that can be applied directly over OBDD structures. The implementation and case studies show the applicability of our method over large state spaces.

We also show that the state explosion problem can be tackled by compositional minimization methods using some suitable equivalence relations. To this aim, we show that two failure based equivalence relations called CFFD and NDFD are congruence relations with respect to product and hiding operators of constraint automata. Therefore, based on the congruency results and because of the linear time temporal logic preservation properties of CFFD and NDFD equivalences and their minimality properties, they can be used for compositional minimization of constraint automata models in the field of model checking. The method is applied on some practical case studies.

1.3 Related Work

Reo [13] is a coordination language based on connectors for the orchestration of components in a component based system. Primitive connectors such as synchronous channels or FIFO queues are composed to build circuit-like component connectors which exhibit complex behavior and play the role of glue code in exogenously coordinating the components to produce a system.

In contrast to many connector languages for components that focus on stateless connectors in a control flow setting (e.g. BIP [33]), Reo generalizes dataflow networks and Kahn networks because it allows to express behavior including state-based, context dependent, multi-party synchronization and mutual exclusion. The original description of Reo was purely informal [13] and no formal semantics for it existed. Subsequently, a number of models were developed to capture the desired behavior of Reo connectors and of their composition. These include models based on constraint automata [30], timed data streams (also known as abstract behavioral types) [27], connector colouring [47], structural operational semantics [116], linear logic [46] and intentional automata [52] . None of these models, however, is entirely satisfactory. Timed data streams model the possible data flow of a network, but because of their declarative nature they have no support for model checking. All other models are more operational and more suitable for analysis techniques, but either they do not give the desired semantics for certain connectors, or they suffer from technical problems such as not being able to give semantics to all connectors, or both.

Constraint automata [30] are acceptors of timed data streams, but are much more concrete and suitable for model checking analysis. A constraint automaton is a labeled transition system in which each transition label contains two parts: a set N of port names that are *synchronized* if the transition is taken and a proposition g on the data. The latter acts as constraint

on data that can be communicated through the ports in N. The data flowing through the ports in N is *mutually exclusive* with respect to any communication by a port not in N.

Two specific shortcomings of modeling Reo by constraint automata, for example, are that it cannot model desired fairness constraints and it cannot model operations that depend upon pending I/O operations on the communication ports of a connector. This latter feature is called *context dependency*, which occurs when the behavior of a connector can change depending upon not only the presence of requests on a connector boundary, but also on their absence. In such cases, the behavior of a connector can change dramatically with changing context. Both connector coloring and Reo automata [37] address the context dependency issue, but connector coloring does not include a description of the temporal unfolding of a Reo connector, and Reo automata do not address fair behaviors. Both models are incomplete in that they cannot give semantics to many reasonable connectors.

Because Reo is one of the most recently proposed coordination languages, there are only a few works on formal verification of the properties of coordination systems specified in Reo. Selecting the formal verification techniques depends on the choice of the formal semantics of Reo. Algorithms for verifying Reo specifications on the basis of their constraint automata semantics have been presented in [30] for checking (bi)simulation and language equivalence and in [17, 18, 45] for temporal logic specifications. In [17, 18] a timed version of constraint automaton was presented and the problem of model checking of timed CTL for it was considered. In [88, 89], timed constraint automata also have been considered as the modeling formalism in presenting a SAT-based approach for bounded model checking of real-time component connectors. The main theme of the work presented in [44, 45] is reasoning about the reconfigurability of Reo networks using CTL like temporal logics and model checking techniques. Also, there is a work on symbolic model checking of a CTL-like temporal logic (called *BTSL* temporal logic) for constraint automata using ordered binary decision diagrams (OBDD) [99, 98]. The implemented tool based on this work is called *Vereofy* [7]. The common features of all of the above mentioned works on verification of Reo networks and constraint automata are: 1- They suppose that all components of the whole system that we need to verify, can be modeled by constraint automata and 2- The temporal logics they use for specification of the properties are branching time. The reconfigurability of Reo networks through algebraic graph transformations and their model checking using the behavioral specification language mCRL2 [61], based on their constraint automata semantics have been considered in [95] and [91, 94].

Compositional verification has been used in a variety of different ways in the analysis of models of concurrency. Clarke et al. in [51] used interface processes to model the environment for a component. They modeled systems as finite transition systems and used CTL to specify their properties. There are some works on compositional verification of systems modeled by I/O automata [106, 117]. Failure based equivalence checking is a technique for compositional verification in which, the goal is to construct a reduced state space that is equivalent to the full state space in the sense of some process-algebraic equivalences (for more theory and references see [139, 140]). There are some experiments on compositional minimization of state spaces using failure based equivalences, such as [105, 86, 141, 142]. One of the main common features of all of these works is that all components of the actual system should be modeled by a general labeled transition system and there is no distinction between the components and the connectors and their properties which we need to verify. For more details

on this method of verification and also some of the experimental results see [140].

There are some tools for describing and then minimizing labeled transition systems, such as the ARA tool set and its most recent version TVT designed in Tampere University of Technology in Finland [151]. One of the most useful tool sets for this purpose is CADP designed in INRIA, France [1]. It contains many useful components for modeling, analysis and minimization of labeled transition systems and it is free for academic use. For the purpose of model checking, there are some tool sets that are specially successfully used in analysis of real systems, like NuSMV [4] and Spin [6]. The NuSMV system is a tool for checking finite state systems against their specifications in the temporal logics LTL and CTL. It uses the symbolic model checking technique on the ordered binary decision diagrams (OBDDs). Spin is a widely distributed software package that supports the formal verification of distributed systems. It uses a high level language for specifying systems descriptions, called Promela, and LTL is its specification language.

1.4 Thesis Outline, Contributions, and Results

This thesis proceeds as follows:

- In the last section of this chapter, we review our own research leading to this thesis. In this way, we introduce the publications on which this thesis is based.

- In chapter 2, we introduce the context and the background of the thesis. We briefly introduce the notions of component based systems and coordination. The problem of formal verification of reactive systems and its solution methods, namely, deductive verification, model checking and their combinations are introduced. Also, the method of automata theoretic model checking and a set of advanced techniques of model checking to deal with the problem of state explosion, including on-the-fly and symbolic model checking and equivalence based compositional reduction, are presented. In addition, we introduce the framework of temporal logics in the context of the linear or branching time views and explain shortly the reason for selecting the linear time view as our logical framework in this thesis.

- In chapter 3, we describe the Reo coordination language and the theory of constraint automata as an operational semantics for Reo that is suitable for model checking. In this chapter, we also briefly describe the other semantic formalisms that have been introduced for Reo, including co-algebraic models, connector coloring, intentional automata, guarded and Reo automata, process algebraic and structural operational semantics.

- In chapter 4, we introduce Büchi automata of records and unconditional fair constraint automata as alternative models for the operational semantics of Reo. We compare their expressiveness with respect to the original model of constraint automata discussed in the previous chapter. In addition, we review some shortcomings of constraint automata and of their timed data streams based semantics in modeling component connectors and

motivate the use of records and Büchi automata of records as operational semantics for Reo:

  - We introduce *records* as data structures for modeling the simultaneous executions of events: ports in the domain of the record are allowed to communicate simultaneously the data assigned to them, while ports not in the domain of the record are blocked so that no communication can happen. The behavior of a network of components is given in terms of (infinite) sequences of records, to specify the order of occurrences of the events. In addition, streams and languages of streams of records are introduced.
  - We give a bidirectional translation of TDS-languages (that are used as the semantics of constraint automata) and record-based languages.
  - The notion of *Büchi automaton of records (BAR)* is introduced and it is shown that each constraint automaton can be translated into a Büchi automaton of records.
  - We show that BAR's can be used to model Reo connectors, in particular connectors with some fairness conditions on their behaviors, using some examples. This proves that BAR's are semantically more expressive than constraint automata.
  - We introduce a join composition operator for Büchi automata on streams of records and show that it is correct with respect to the join operator for constraint automata.
  - Also, we present a method to recast the join operation on BAR's using the standard product operator of Büchi automata.
  - We introduce a more expressive version of constraint automaton, called *fair constraint automaton*, whose syntax is the same as constraint automaton but now with final (accepting) states and its semantics is based on the languages of streams of records.

- In chapter 5, in order to address context-dependent behaviors, we extend our BAR models with the possibility of testing if some ports of the environment are ready to communicate or not. That is, we consider a Büchi variant of Kozen's finite automata on guarded strings [100] which we call *augmented Büchi automata of records (ABAR)*. Our model has an advantage over previous models in that it covers the basic concepts of Reo as well as the context sensitive behavior within a standard automata theoretical framework. The benefits are a clear and easy notation for the representation of a component connector, as well as efficient existing tool support for automatic analysis. The chapter introduces the following notions and results:

  - We introduce augmented Büchi automata of records (ABAR) as acceptors of infinite guarded strings of records.
  - We show that in addition to the fairness constraints, the context-dependent behavior of Reo connectors can be modeled using ABAR.
  - We introduce a join composition operator for augmented Büchi automata on streams of records.

  - Also, we present a method to recast the join operation on ABAR's using the standard product operator of Büchi automata.
  - We introduce a context dependent version of fair constraint automaton, called *augmented fair constraint automaton*, with the same syntax as constraint automaton but now it has final (accepting) states and labels on states and with a semantics based on languages of infinite guarded strings of records.

- In chapter 6, we present an automata theoretic method of model checking for coordination systems modeled by ABAR's. For this aim, we follow the following steps:
  - First, we introduce an action (or transition) based linear temporal logic (called ρLTL) interpreted over computations of ABAR's.
  - Then, we show that ρLTL formulas can be translated into ABAR's using an inductive translation method.
  - Also, we present an on-the-fly method to translate ρLTL formulas into ABAR's.

- In chapter 7, we introduce the main theoretical and practical concepts we used to implement a BDD based model checking tool for Reo specifications. This implementation is based on the augmented Büchi automata of records semantic models introduced in the previous chapters. Moreover, this tool accepts properties expressed in the ρLTL linear temporal logic as input and verifies the Reo specification against these properties. The chapter also presents some case studies.

- In chapter 8, we investigate the method of *equivalence based compositional minimization* of models of Reo to deal with the state explosion problem in model checking. In this method components of a system are reduced with respect to an equivalence relation before building the complete system [60, 50, 79, 82]. An equivalence relation should have two properties in order to be useful in the equivalence based compositional reduction method: it should *preserve* the class of properties to be verified and it should be a *congruence* with respect to the syntactic operators that are used to compose the components of the model. By congruence relation we mean that the replacement of a component of a model by an equivalent one should always yield a model that is equivalent with the original one. Fortunately, in the context of compositional failure based semantic models of process description languages such as CCS and LOTOS, there are two equivalence relations, called CFFD and NDFD [86, 141, 142], that have a significant property: CFFD-equivalence preserves the fragment of linear time temporal logic that has no next-time operator and has an extra operator for distinguishing deadlocks [141, 142]; and NDFD-equivalence preserves linear time temporal logic without next-time operator [86]. It was also shown that CFFD and NDFD are the minimal equivalences preserving the above mentioned fragments of linear time temporal logic with respect to all composition operators of CCS and LOTOS [86]. Thus, if we use CCS-like composition operators, CFFD and NDFD equivalences can be suitable equivalences for using in the context of equivalence based compositional reduction method.

  In chapter 8, we investigate the above mentioned results for the case of constraint automata. In other words, we consider the failure based semantics for constraint automata

as labeled transition systems with compound labels, instead of their timed data streams-based semantics. Thus, we follow the following steps:

  - First we define CFFD and NDFD equivalences for the case of constraint automata.
  - We show that the temporal logic preservation results hold in the case of constraint automata.
  - We consider the congruency results and prove that:
    * The failure-based equivalence relation CFFD is a congruence with respect to the join operator of constraint automata.
    * The failure-based equivalence relation CFFD is a congruence with respect to the hiding operator of constraint automata.
    * The failure-based equivalence relation NDFD is a congruence with respect to the join operator of constraint automata.
    * The failure-based equivalence relation NDFD is a congruence with respect to the hiding operator of constraint automata.
  - We show that the minimality properties of CFFD and NDFD also hold in the case of constraint automata, that is, CFFD and NDFD are the minimal equivalences preserving the above mentioned fragments of linear time temporal logic with respect to the composition operators of constraint automata.
  - Then, we introduce the compositional model checking method for component based systems whose coordination subsystem and the interfaces of all components are modeled by constraint automata.
  - We introduce an implementation of the above mentioned method of model minimization and show its usefulness in practice by summarizing the results of some case studies.

- In chapter 9, we summarize our ideas and results and also suggest some research directions that can be considered as future work based on this thesis.

1.5 Research History and Publications

As mentioned before, the goal of this thesis is to find a *formal verification* framework for component based systems in general and for their coordination subsystems in particular. We have chosen *model checking of temporal logic properties* as the verification method. From a historical point of view, our research can be divided into four periods or steps:

- In the first step, our focus was on working on constraint automata as the operational semantics of coordination systems and as the models of the interfaces of components. Thus, in this phase, our aim was to prepare a compositional and hierarchal environment for model checking of linear time properties of constraint automata models in

the presence of the state explosion problem. Our main ideas to do this were presented in [82, 77]. To tackle the state explosion problem, we investigated two different solution methods:

  - *Compositional minimization* of models using suitable equivalence relations. For this aim, in [79, 78] we introduced two failure-based equivalence relations CFFD and NDFD as new semantics for constraint automata and our proposal for using them to reduce the size of models for model checking. We also proved that CFFD and NDFD are congruences with respect to join and hiding composition operators of constraint automata [75, 74]. Moreover, we showed that the temporal logics preservation and minimality properties of CFFD and NDFD hold for the case of constraint automata [76]. Based on these results, we introduced a method for compositional model checking of component based systems, reduction algorithms, and their implementations and application to case studies [76, 69].
  - As another way to deal with the state explosion, we considered *abstraction* techniques. In this method, using some suitable mapping from a large state space into a smaller one that preserves desired sets of linear temporal properties, we try to do model checking over the smaller model. A suitable mapping can be obtained by reducing the number of state variables from the model whose state variables are more to another one with less variables. Suitable mappings preserve linear time fairness and liveness properties [69, 119, 118].

- In the second step, based on the shortcomings of constraint automata in modeling all aspects of the behavior of connectors and the complication of their timed data streams based semantics, we focused on presenting a new operational semantics for Reo and on their model checking. Our main motivation in this phase was to use the classical theory of Büchi automata and their model checking using translation of linear temporal logics into automata [71, 73, 38, 72]. This step consisted of three phases:

  - In the first phase, we focus on the shortcomings of constraint automata in modeling some fairness constraint over the behavior of connectors and also their complicated timed data streams based semantics [71, 72]. To solve these problems:
    * We introduced *records* as data structures for modeling the simultaneous executions of events: ports in the domain of the record are allowed to communicate simultaneously the data assigned to them, while ports not in the domain of the record are blocked so that no communication can happen. The behavior of a network of components is specified in terms of (infinite) sequences of records, which give the order of occurrences of the events.
    * Also, we introduced *Büchi automata of streams of records* (BAR) as the operational semantics of coordination systems that cover the synchronization of the I/O operations and several fairness constraints over the behaviors of channels.
    * We obtained a main result that every constraint automaton can be translated into an essentially equivalent Büchi automaton of records.

  - In the second phase, we focused on the shortcomings of constraint automata in modeling the context dependent behaviors of some connectors. We introduced *augmented Büchi automata of records* (ABAR) which extend our BAR model with the possibility of testing if some ports of the environment are ready to communicate or not. ABAR's are defined as acceptors of infinite guarded strings of records. We also introduced a composition operator for ABAR's and showed its correctness with respect to the intended semantics by means of several examples. Finally, we showed that our composition operator can be decomposed into two operators: record extension and ordinary automata product [73, 72].
  - In the third phase, we intended to do model checking of linear time properties of ABAR models [38]. To this aim:
    * We defined an action based linear time temporal logic for expressing properties of Reo connectors, called ρLTL.
    * We showed that ρLTL formulas can be synthesized into Büchi automata representing Reo connectors, thus leading to an automata based model checking algorithm.
    * By generalizing standard automata based model checking algorithms for linear time temporal logic, we gave both global (inductive) and on-the-fly algorithms for the model checking of formulas for Reo connectors.

- In the third step, we implemented a tool set for model checking of Reo specifications using all of the above mentioned results. Now, we have two implemented tools designed independently based on the results of the two above steps. The first one, called *ArQuVer* (Architecture Quality Verification tool), is a tool that is intended to prepare an environment for specification of software architectures using constraint automata and verification of their properties, especially nonfunctional and qualitative properties of software architectures. It contains components for minimization of constraint automata using bisimulation, trace, CFFD and NDFD equivalence relations. The other, is a tool for model checking of ρLTL formulas over ABAR models. It implements the models using BDD data structure and checks the formulas directly over the BDD's. As a future work, we will incorporate these tools into an integrated one. Some results of the first tool were reported in [76, 69, 74]. For the other one, the first paper reporting our results is currently under review.

- The fourth step which is our intended future work, is to extend our theoretical and practical results into real-time and probabilistic coordination systems, using timed and probabilistic extensions of our BAR and ABAR models. Recently, we have presented a set of results on timed Büchi automata of records and the model checking of the linear time properties [8].

2 Context and Backgrounds

In this chapter, we introduce the context and the background of the thesis. In Section 2.1, we briefly review the notions of component based systems and coordination. The problem of formal verification of reactive systems and its solution methods, namely, deductive verification, model checking and their combinations are discussed in Section 2.2. Section 2.3 is an overview of a set of advanced techniques for model checking, used in this thesis to deal with the problem of state explosion. In Section 2.4, we describe the framework of temporal logics in the context of the linear or branching time views and explain briefly the reason for selecting the linear time view as our logical framework in this thesis.

2.1 Component Based Systems and Coordination

The concept of *component based systems*, especially component-based software, is a new philosophy or way of thinking to deal with the complexity in designing large scale computing systems[13, 135]. One of the main goals of this approach is to compose reusable components by some glue code. The model or the way in which these *components* are composed is called a *coordination model*. Thus, a component-based system has two main parts: a set of components and a coordinating subsystem.

What are Components? A *component* is defined as a unit of computation that can be used independently and is subject to composition by a third party [135]. To be used independently, a clear distinction of the component from its environment and the other components is required. Thus, the component communicates with its environment through its *interface*. The interface of a component can be defined as a specification of its access points [135]. Therefore, a component is a black box computing system that communicates with the other components and its environment through its interface by which it offers its provided services.

What is Coordination? A component based system needs connections and interactions among its components. The main goal of *coordination* is to manage the interaction among the components. Thus, coordination can be defined as the study of the dynamic topologies of interactions among components, and the construction of protocols to realize such topologies that ensure well-behavedness [15].

The task of coordination can be realized using different ways of communication, management and control strategies, methods of creation and termination of computational activities, and ways of synchronization of components. A set of definite choices for all of these different aspects is referred as a *coordination model*.

Furthermore, a *coordination language* is the linguistic formalism that is used to specify a coordination model. Thus, a coordination language offers facilities for the specification of the ways of controlling synchronization, communication, creation and termination of computational activities, and other aspects of the coordination model.

There are many languages and models for coordination which provide formal description of the glue code for plugging components together. Coordination models and languages can be classified in different categories using (at least) three different aspects [15]:

- *Dataflow-oriented* versus *Control-oriented* and *Data-oriented* Coordination. The task of coordination can be realized by focusing on data, control or the dataflow in the component based system. In other words, the coordinator can enforce its management strategies over the whole system by focusing on the shared body of data and its evolution or on processing and flow of control. The flow of control can generally be considered not only over the flow of data but also over the control points of components. However, the flow of control can in particular, be considered only over the flow of data. Coordination models that realize their strategies by focusing on data or on the flow of control are called *data-oriented* and *control-oriented* coordination, respectively. In particular, those that coordinate the system only by focusing on the flow of data are called *dataflow-oriented* [15, 120, 12]. For instance, Linda [9, 41, 42] uses a data-oriented coordination model, whereas Manifold [24] is a control-oriented and Reo [13] is a dataflow-oriented coordination language.

- *Exogenous* versus *Endogenous* Coordination. A coordination model can consider the components that are under its coordination as open or black-boxes. A coordination model is called *exogenous* if its view about the components is that they are black-boxes that the coordinator has no access to their internal structure and they should be coordinated from outside. In contrast, an *endogenous* coordination model provides a set of primitives that must be incorporated *within* the components [15]. For instance, Linda [9, 41, 42] is based on an endogenous model, whereas Manifold [24] and Reo [13] are exogenous coordination languages.

- Coordination Mechanisms. Based on the mechanisms of coordination and the mediums through which the coordinator enforces its strategies, coordination models and languages can be classified in the four categories [133, 15]:

  - Coordination through *massage passing*. In this type of coordination, components send messages to each other. The connections between components can be defined (1) as a point-to-point model where each massage has a specific origin and target, or (2) as a publish-and-subscribe model where a component can send a message meant for any component having some kind of a specific service.
  - *Event-based* coordination. In this mechanism, a component, called the producer or event source, can create and fire events, the events are then received by other components, called consumers or event listeners, that listen to this particular kind of events [133].
  - Coordination through *shared data spaces*. In this mechanism of coordination, there is a set of shared data spaces that all components can read from or write values to with specific formats, usually tuples like in Linda [9, 41, 42]. This shared data values contain both data and condition (control) fields. The coordination task is realized by the predefined protocol of reading and writing over the shared data spaces by the components.
  - *Channel-based* coordination. A channel is a one-to-one connection that offers two ends, its source and its sink, to components. A component can write by inserting values to the source-end, and read by removing values from the sink-end

of a channel; the data exchange is locally one way: from a component into a channel or from a channel into a component. The communication is anonymous: the components do not know each other, just the channel-ends they have access to [133, 15]. Reo is an instance of a specification language for the channel based coordination models [13].

There are some more formal and mathematical specification or modeling formalisms for coordination, such as interface automata [10], I/O automata [106], and all of the operational models introduced for Reo [13], namely constraint automata [30], our BAR and ABAR models [71, 73, 72], timed constraint automata [17, 18], probabilistic constraint automata [28], stochastic constraint automata [31], intentional constraint automata [52], Reo automata [36, 37], and port automata [90, 96]. In a more fundamental view, each of these formal models by itself can be used as a formalism for modeling coordination systems. Also, according to a high level and very abstract view, all of these formalisms are fundamentally labeled transition systems with some constraints, guards or classifications on their transitions. But their expressive powers for modeling of coordination systems and also, in practice, their efficiency as the specification tools in the context of verification and model checking are different.

Reo is one of the most recent proposed coordination models. (The first paper on Reo was published in 2002 [25].) Reo is a channel-based exogenous coordination language in which complex coordinators are built out of simpler ones [13]. Reo is a dataflow-oriented channel based exogenous coordination model, in which, the glue code is provided by a network of channels obtained through a series of operations that create channel instances from a set of primitive channels and link them in the network of *nodes* [13]. By Reo specifications one can specify the coordinating model in a compositional and hierarchal way. We describe Reo and constraint automata in chapter 3.

2.2 Formal Verification and Its Methods

The main goal of verification methods is to ascertain that an actual system or program satisfies its requirements. In *formal verification* methods one tries to achieve the aim of verification by describing the system using a mathematical model, expressing the requirements as properties of this model and by showing through rigorous mathematical reasoning that the model of the system indeed has the required properties [50, 110]. Thus, the techniques for formal verification can be considered as comprising of three parts [68]:

- a *framework for modeling systems*, which is typically a description language or some sort of a transition system;
- a *specification language* for describing the properties to be verified;
- a *verification method* to establish whether the description of a system satisfies the specification.

From the view point of mathematical logic, if the desired property to be verified is expressed by a formula in a formal language, say ϕ, the process of reasoning about the correctness of ϕ can be done using *proof theoretic* or *model theoretic* methods or a combination of them. In the literature of formal verification, using proof-based methods is called *deductive verification*, while using the model theoretic approach is called *model checking*. In the following subsections, we briefly describe these methods of formal verification and the ways in which they are combined.

2.2.1 Deductive Verification

In *deductive verification*, the system intended to be proved correct is described by a set of formulas called *axioms* and the correctness property is expressed by a formula in the same language of axioms. The process of verification is to derive a proof of the desired correctness property based on the set of axioms. In other words, if the property to be verified, say ϕ, is expressed in a formal language with a well-established proof theory, and the system is specified by a set of formulas Γ, the deductive verification method is to try to find a proof for $\Gamma \vdash \phi$.

Obviously, if we are interested to do the deductive verification automatically, we need to have an implemented automatic theorem prover for the formal language in which the axioms and the desired property are expressed. Typically, it is supposed that an expert (mathematician or logician) uses the theorem prover and helps it to find the best ways of doing the proof. The facts that only experts can use a theorem prover and fully automated theorem provers are not found are of the main disadvantages of the method of deductive verification. Another problem with theorem proving is that it is not particulary suitable in providing *debugging information*, that is, information on the nature and location of errors [139].

An advantage of deductive verification is that it can be used for reasoning about infinite state systems such as systems that work on unbounded data types (for instance the ordinary integers). As another example, theorem proving techniques work well when process structures evolve, that is, new processes can be added and old processes can be aborted during the executions of the system [139]. In these cases, the task of deductive verification can be automated to a limited extent. However, even if the property to be verified is true, no limit can be placed on the amount of time or memory that may be needed in order to find a proof [50].

2.2.2 Model Checking

If the correctness requirements of a formally modeled computing system are given in a mathematical notion, such as linear temporal logic [110], branching time temporal logic [148] or automata on infinite objects [138], an algorithmic model theoretic process called *model checking* [50] can be used to check if the system respects its correctness requirements. In other words, if the property to be verified, say ϕ, is expressed in a formal language L and the system is modeled by a formal model $\mathcal{M}$ (based on the formal semantics of L) the model checking method is to check whether $\mathcal{M} \models \phi$. In brief, the process of model checking a desired property of a system consists of the following three steps:

- *Modeling* all of the possible states and behaviors of the system using a suitable formalism. Typically some sort of finite transition systems such as finite automata over finite

strings or trees, finite automata over infinite words or trees (such as Büchi automata), finite Petri nets, finite Markov chains and so on are used as the modeling formalism. Some model checkers accept the description of the system in some more high-level description languages such as programming languages, say C or Java, or hardware description languages such as Verilog or VHDL, and automatically convert them into the intended transition systems. Obviously, based on the limitations of time and memory or the interesting aspects of the system, each kind of modeling abstracts away several aspects of a real system.

- *Specification* of the property to be verified in a suitable language. Typically some sort of *temporal logics* or *automata-based expressions* are used for the specification of desired properties. There are several kinds of temporal logic in computer science. We discuss the rules of temporal logic proposed in model checking in Section 2.4.

- Running the *verification* algorithm to check whether the model of a system satisfies its specification explores all possible system states in a brute-force manner. Ideally this algorithm is executed completely automatically. In case of a negative result, the user is often provided with an error trace. This can be used as a *counterexample* for the checked property and can help the designer in tracking down where the error occurred. In this case, analyzing the error trace may require a modification to the system and reapplication of the model checking algorithm.

In comparison with the other approaches to verification, the model checking method enjoys some remarkable advantages: 1- It is fully automatic, and its application requires no user supervision or expertise in mathematical disciplines such as logic and theorem proving. 2- When a design fails to satisfy a desired property, the process of model checking always produces a counterexample that demonstrates a behavior which falsifies the property. This faulty trace provides an insight to understanding the real reason for the failure. 3- In comparison with theorem proving, model checking techniques are highly flexible: a large set of analysis and verification questions of many different types can be answered using a single state space based model.

Model checking has shown to be an efficient and easy to use technique in computer systems verification. However, there is a major drawback in using model checking: the model of real system can be extremely large. In the literature this problem is often referred as the *state explosion problem*. In Section 2.3, we describe some techniques that have been suggested to deal with this problem. Another disadvantages of model checking is that it does not work for infinite-state systems.

2.2.3 Combining Deduction and Model Checking

The main advantage of theorem proving is its ability to reason about infinite state spaces and the main benefit of model checking is its ability to be implemented fully as an automatic tool. They can be composed to yield advanced techniques for verification such as abstraction and compositional reasoning.

As a way to deal with the problem of state explosion, *abstraction* techniques have been proposed in order to integrate theorem proving and model checking [49, 104, 102]. The idea

is to reduce the original system to a smaller model via interactive proof techniques. In a second step, the smaller system is analyzed using automatic model checking tools. Usually, the smaller system is obtained by partitioning the original state space via a structure-preserving function between the two state spaces called *abstracting function*. The theorem prover is employed in order to guarantee the abstraction to be sound, that is if the abstract system satisfies a property, so does the original system [117].

Another way to overcome the state explosion problem is *compositional reasoning* techniques: decomposing the process of reasoning about the behavior of a composed system into partial reasonings about the behaviors of its constituents. For this purpose, the system and the desired property both are decomposed into components and local properties. The correctness of each local property of a component is verified by model checking, while showing that the correctness of local properties entails the correctness of the whole desired property is done by theorem proving [86, 123].

2.3 Advanced Techniques of Model Checking

As mentioned before, state explosion is the main problem in model checking. The state space of the model of a program or hardware system can be very large, even for very simple cases such as a program that performs a simple computation over integers.

During the last two decades, many methods have been suggested to reduce the number of states that must be constructed for answering certain verification or analysis questions. Such enhanced state space methods increase substantially the size of the systems that can be verified, while preserving most of the advantages of the model checking method of verification. Equivalence based compositional model checking [86, 123], partial order reduction by representatives [121], the pre-order reduction techniques [60], abstraction methods [49, 104], using the symmetric structure of the models [56], automata theoretic model checking [145, 146, 102] and its on-the-fly enhancements [59], and symbolic model checking [111, 50] are the most important methods for dealing with the state explosion problem. In the following subsections, we describe four model checking techniques which we will in this thesis.

2.3.1 Automata Theoretic Model Checking

Finite automata can be used to model concurrent and interactive systems. Either the set of states or the alphabet set can then represent the states of the modeled system. One of the main advantages of using automata for model checking is that both the modeled system and the specification are represented in the same way. If $\mathcal{A}$ is an automaton modeling the actual system, let $\mathcal{L}(\mathcal{A})$ be the set of all possible behaviors of the system. The specification of the desired property can also be given as an automaton $\mathcal{S}$, over the same alphabet. Then, $\mathcal{L}(\mathcal{S})$ is the set of all allowed behaviors.

Now, the system modeled by $\mathcal{A}$ satisfies the specification $\mathcal{S}$ when $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S})$ which is equivalent to $\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset$. This means that there is no behavior of $\mathcal{A}$ that is disallowed

by $\mathcal{S}$. If the intersection is not empty, any behavior in it corresponds to a counterexample. Thus, the process of automata theoretic model checking can be summarized by the following steps:

- 1- *Modeling* all of the possible states and behaviors of the system using a finite automaton, say $\mathcal{A}$, over finite or infinite words or trees (such as Büchi automata).
- 2- *Specification* of the property to be verified using an automaton, say $\mathcal{S}$. Then, complementing the automaton $\mathcal{S}$, that is, constructing an automaton $\overline{\mathcal{S}}$ that recognizes the language $\overline{\mathcal{L}(\mathcal{S})}$.
- 3- Constructing the *product automaton* that accepts the intersection of the languages of $\mathcal{A}$ and $\overline{\mathcal{S}}$.
- 4- Checking the emptiness of the language of the product automaton. If this language is empty, we announce that the system satisfies the property. Otherwise, every member of the language of the product automaton is a counterexample that can be considered in the debugging of the system.

Moreover, the automaton $\mathcal{S}$ may be obtained using a translation from some specification language such as linear time temporal logic (LTL). In this case, instead of translating a property ϕ into $\mathcal{S}$ and then complementing $\mathcal{S}$, we can simply translate $\neg\phi$ which immediately provides an automaton for the complement language. In this case, the second step (specification step) of the above method is replaced by the following ones:

- 2-1 *Specification* of the property to be verified using a temporal logic formula ϕ.
- 2-2 *Translation* of the formula $\neg\phi$ into an equivalent automaton called $\overline{\mathcal{S}}$.

Thus, the process of automata theoretic model checking in the context of temporal logic specifications is crucially dependent on the following factors:

1- The existence of a translation method from the selected temporal logic into the selected automata formalism.

2- The decidability and complexity of the algorithm to produce the product automaton that accepts the intersection of the languages of two automata of the selected automata formalism.

3- The decidability and complexity of the algorithm to check the emptiness of the language of an automaton of the selected automata formalism.

Fortunately, for the cases of linear time temporal logic (LTL) and Büchi automata, which we will use in this thesis, all of the above three constructions can be done effectively.

2.3.2 On-the-Fly Model Checking

In on-the-fly model checking the algorithm that checks the correctness of a property is integrated into the algorithm that constructs the state space of the system's model. The construction of the state space is immediately stopped when an error against the property is found. Thus, in general, an on-the-fly method speeds up the model checking process in the cases of incorrect systems while it does not for correct systems. Also, this method may reduce the

number of states in another way, by avoiding the construction of those parts of the state space that are not relevant for the property [139].

On-the-fly verification methods can often be combined with other advanced techniques for model checking. For instance, its combination with automata theoretic technique can be explained as follows. Let $\mathcal{A}$ be the Büchi automaton model of a system and $\bar{\mathcal{S}}$ be another automaton describes the negation of the desired property. In the automata theoretic model checking the emptiness of the intersection of $\mathcal{A}$ and $\bar{\mathcal{S}}$ is checked. If the intersection is not empty, a counterexample is reported. Using an on-the-fly method, instead of constructing the automata for both $\mathcal{A}$ and $\mathcal{S}$ first, we construct only the property automaton $\mathcal{S}$. We then use it to guide the construction of the system automaton $\mathcal{A}$ while computing the intersection. In this way, we may frequently construct only a small portion of the state space before we find a counterexample to the property being checked [50]. On the other hand, sometimes it is also possible that the property automaton $\mathcal{S}$, which is obtained in a translation process from a temporal logic formula, itself can be constructed step-by-step using an on-the-fly translation method [59, 107]. Considering both cases at the same time, we obtain a model checking procedure where the product (intersection) automaton is constructed on-the-fly (by constructing both the system and the property automata on-the-fly), during a depth-first search which checks for emptiness.

2.3.3 Symbolic Model Checking

The model-checking procedure described so far relies on the assumption that the transition system has an explicit representation by the predecessor and successor lists per state. Such an enumerative representation is not adequate for very large transition systems. To counter the state explosion problem, the model-checking procedure can be reformulated in a symbolic way where sets of states and sets of transitions are represented rather than single states and transitions [111, 29]. More generally, we can call this approach to deal with the state explosion problem the method of using *packed state spaces* [139].

There are several possibilities to realize model checking algorithms in a packed state space stetting. The most prominent ones rely on a binary encoding of the states, which permits identifying subsets of the state space and the transition relation with switching functions. To obtain compact representations of switching functions, special data structures have been developed, such as *ordered binary decision diagrams* (OBDD) [111, 50, 29]. In this method, the state set, the transition relation over the set of states and the labeling function which assigns proposition sets onto states, all are encoded by OBDD models. Then the verification algorithms and tools have to be modified to work on the packed state spaces instead of ordinary ones.

The efficiency and usefulness of the symbolic representation of the state space using OBDD models and their suitable verification algorithms are crucially dependent on the selected ordering over the set of boolean variables, which is fixed before constructing the OBDD models. It can be shown that for some switching function, using two different variable orderings produces OBDD structures whose sizes differ exponentially [29]. Since for many switching functions the OBDD sizes for different variable ordering can vary enormously, the efficiency of OBDD-based computations crucially relies on the use of techniques that improve a given variable ordering. However, the problem to find an optimal variable ordering is

known to be computationally NP-hard [34]. Thus, the main problem with the symbolic model checking method is to find the best variable ordering.

2.3.4 Compositional Minimization

Compositional model checking is one of the proposed methods for dealing with the problem of state explosion [50, 51, 123]. In the compositional verification of a system, one seeks to verify properties of the system using the properties of its constituent modules. In general, compositional verification may be exploited more effectively when the model is naturally decomposable [128]. In particular, a model consisting of inherently independent modules is suitable for compositional verification. A special case of compositional verification is the method of *equivalence based compositional reduction*. In this method components of a system are reduced with respect to an *equivalence relation* before building the complete system [60, 50, 79, 82].

In order to be useful in model checking, the selected equivalence relation should satisfy two properties: *preservation* of all properties to be verified and being a *congruence* relation with respect to all operators that are used for composing the models. By a congruence relation we mean that the replacement of a component of a model by an equivalent one should always yield a model which is equivalent to the original one. Thus, two main criteria for selecting an equivalence relation to be used for reducing the models are the set of properties to be verified and the composition operators that are used over the models. If using of an equivalence relation to reduce the size of models produces the smallest ones, we refer to the reduction procedure as *minimization*.

For example, in the context of failure based semantic models of the process description language LOTOS, there are two equivalence relations, called *chaos-free failures divergences (CFFD)* and *non-divergent failure divergences (NDFD)*, which satisfy the preservation property for two fragments of linear temporal logic. NDFD preserves all properties that are expressible in linear time temporal logic without next-time operator (called LTL_{-X}) [86]. Also, CFFD preserves all properties that are expressible in linear temporal logic without the next-time operator but with an extra operator that distinguishes deadlocks from divergences (called LTL^{ω}) [141, 142]. Also, it has been shown that CFFD and NDFD are the weakest equivalence relations that preserve the above mentioned fragments of linear temporal logic. In other words, they both produce the minimized transition systems with respect to the preserved temporal logics. Moreover, it has been shown that in the case of standard labeled transition systems CFFD and NDFD are congruences with respect to all composition operators defined in LOTOS [142].

2.4 The Rules of Temporal Logics

To specify and then verify the desired properties of computer and computing systems, we need a logic for specifying properties of the state-transition models. Classical propositional and predicate logics allow to build up complicated expressions describing properties of states.

For *reactive systems*, correctness depends on the executions of the system, not only on (the state of) the input and output of a computation and on fairness issues. *Temporal logic* is a formalism for treating execution path and fairness aspects. Temporal logic extends propositional or predicate logic by modalities that permit to refer to the infinite behavior of a reactive system. They provide a very intuitive but mathematically precise notation for expressing properties of the relation between the state labels in executions.

The underlying nature of time in temporal logics can be either linear or branching. In the linear view, at each moment in time there is a single successor moment, whereas in the branching view it has a branching, tree-like structure, where time may split into alternative courses. Based on which view is chosen, a system of temporal logic is classified as either a linear time temporal logic in which the semantics of time structure is linear, or a branching time logic corresponding to branching time structures as the semantics of the formulas.

The most popular linear time temporal logic that is used in the field of model checking is the propositional version of a temporal logic called *LTL (Linear Temporal Logic)*. In the linear temporal logic LTL, formulas are composed from the set of atomic propositions using the usual Boolean connectives as well as the temporal connective $\Box$ (always), $\Diamond$ (eventually), $\bigcirc$ (next), and U (until).

On the other hand, the most popular branching time temporal logic in the field is a propositional branching time temporal logic called *CTL (Computational Tree Logic)*. Also, there is a more expressive branching time temporal logic that is called *CTL** [55]. The branching temporal logic CTL* extends LTL by the path quantifiers E (there exists a computation) and A (for all computations). The branching temporal logic CTL is a fragment of CTL* in which every temporal connective is preceded by a path quantifier. CTL* is more expressive than both temporal logics CTL and LTL. The expressiveness of CTL is not comparable to that of LTL. For example, the LTL formula $\Diamond\Box p$ is not expressible in CTL, while the CTL formula $A\Diamond A\Box p$ is not expressible in LTL. There are several other temporal logics such as the branching temporal logic $\forall$CTL that is a fragment of CTL in which only universal path quantification is allowed and linear time or branching time μ-calculus.

The discussion of the relative merits of linear versus branching temporal logics in the context of specification and verification goes back to 1980 [148, 147]. There are several papers arguing against or in favor of linear and branching views (for more see [147]). Our choice in this thesis is the *linear time* approach, not because of such kinds of arguments. The main reason for our choice is because we will study *automata theoretic* and *compositional minimization* methods for which linear time temporal logics are more applicable [147, 148, 146, 143, 139].

3 Formal Modeling of Component Connectors

Reo is an exogenous coordination language for compositional construction of coordination systems. Constraint automaton has been defined as the operational semantics of Reo. In the first two sections of this chapter, we describe Reo and constraint automata. In the third section, we briefly present the other semantic formalisms that have been introduced for Reo.

3.1 Reo: A Channel Based Coordination Language

Reo is a coordination language which can model and specify coordination of a set of components trough networks of channels or compositional connector built out of primitive channels. In this section, we introduce Reo primitives and compositional coordination constructions which can be obtained by using it as introduced in [13, 14, 19, 25].

3.1.1 Reo Primitives

Reo is a coordination language which is based on a calculus of channels [13, 14, 19, 30]. By using Reo specifications, complex component connectors can be organized in a network of channels and build in a compositional manner. The simplest connectors in Reo are a set of *channels* with well-defined behavior supplied by users. Reo can be used as a coordination language for concurrent processes or as a "glue language" for compositional construction of connectors that orchestrate component instances in a component based system. The emphasis in Reo is on connectors and their composition only, not on the entities that connect to, communicate and cooperate through these connectors.

Reo uses a simple notion of channels and can model any kind of peer-to-peer communication. The only requirements for a channel used in a Reo network are that the channel should have two channel ends, called as *sink* or *source* ends, and a well-defined semantics which constraints or relates the flow of data through these ends. At a source end data items enter the channel by performing corresponding write operations. Data items are received from a channel at its sink end by performing corresponding read operations. Reo allows for an open ended set of channel types with user defined semantics. Some primitive channels relevant for this thesis are shown in Figure 3.1 by their graphical representations.

Every synchronous or FIFO channel has a source and a sink end. A *synchronous channel* (abbreviated by *Sync*) has no buffer and accepts a data item through its source end if and only if it can simultaneously dispense it through its sink. A *FIFO1 channel* is represented graphically by a small box in the middle of an arrow. Writing a data item at the source end of a FIFO1 is enabled as long as the buffer is empty. The effect of writing d is that d will be stored in the buffer. Reading at the sink end is enabled if the buffer is full, in which case the data item is taken off from the buffer. FIFO channels with two or more buffer cells can be produced by composing several FIFO1 channels [30].

A *lossy synchronous channel* (abbreviated as *LossySync*) is similar to synchronous channel, except that it always accepts all data items through its source end. If it is possible for it to simultaneously dispense the data item through its sink (e.g. there is a take operation pending on its sink) the channel transfers the data item, otherwise the data item is lost. For a

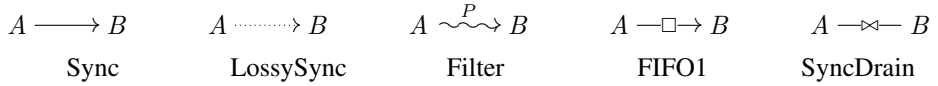


Figure 3.1: Some useful channel-types in Reo

synchronous filter channel, its pattern P (for our purpose here, formalized as a subset P of data set $\mathcal{D}$) specifies the type of data items that can be transmitted through the channel. Any value $d \in P$ is accepted through its source end iff its sink end can simultaneously dispense d. All data items $d \notin P$ are always accepted through the source end but are immediately lost. The *P-producer* is a variant of a synchronous channel whose source end accepts any data item $d \in \mathcal{D}$, but the value dispensed through its sink end is always a data element $d \in P$.

Two very useful channels for the design of complex coordination principles in Reo are the *synchronous and asynchronous drains*. Because a drain has no sink end, no data value can ever be obtained from these channels. Thus, a synchronous drain accepts a data item through one of its ends iff a data item is also available for it to simultaneously accept through the other end as well. All data accepted by this channel are lost. An asynchronous drain accepts and loses data items through its two source ends, but never simultaneously. *synchronous and asynchronous spout* are duals of their corresponding drain channel types, as they have two sink ends.

3.1.2 Compositional Connectors

Every channel represents a simple connector with two ends. More complex connectors are constructed in Reo out of the simpler ones using its *join* operation. Joining of two connectors is plugging their ends together into nodes such that the data exchanged through the ports that coincide on the same node are synchronized, but the I/O behaviors of the other ports remain as they were before.

Reo defines a ***connector*** as a set of channel ends and their connecting channels organized in a graph of ***nodes*** and ***edges*** such that:

- Zero or more channel ends coincide on every node.

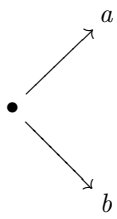
- Every channel end coincides on exactly one node.

- There is an edge between two (not necessarily distinct) nodes if and only if there is a channel end which coincides on each of those nodes.

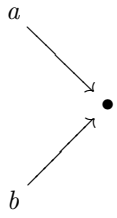
Let x be a channel end and N be a node. We use $x \mapsto N$ to denote that x coincides on N, and $\hat{x}$ to denote the unique node on which the channel end x coincides. For a node N, the set $[N] = \{x | x \mapsto N\}$ is the set of all channel ends coincide on N and is partitioned into the disjoint sets $Src(N)$ and $Snk(N)$, denoting the sets of source and sink channel ends that coincide on N, respectively. The nodes of a Reo network represent sets of channel ends. They arise through Reo's *join* operator and can be classified into *source, sink* and *mixed* nodes, depending on whether all channel ends that coincide on a node N are source ends (then N is a source node), sink ends (then N is a sink node) or whether N combines sink and source ends (then N is a mixed node). Source and sink nodes represent input and output ports where components might connect to the network. The mixed nodes serve as routers where

data items can be transmitted through the network. More formally, a node N is a ***source node*** if $Scr(N) \neq \emptyset \wedge Snk(N) = \emptyset$. Analogously, N is a ***sink node*** if $Snk(N) \neq \emptyset \wedge Scr(N) = \emptyset$. A node N is a ***mixed node*** if $Scr(N) \neq \emptyset \wedge Snk(N) \neq \emptyset$.

Components are connected only to sink or source nodes, they cannot connect to mixed nodes. At most one component can be connected to a source or a sink node. A source node acts as a *replicator*, while a sink node acts as a nondeterministic *merger*. Consider the following source node:



A write operation succeeds on the node if the source ends a and b are both ready to accept the data item, in which case it is written to both source ends. Thus, the data item is replicated. On the other hand, take the following sink node:



A read or take operation succeeds on the node only if at least one of the sink ends a or b is ready to offer a suitable data item into the node. If both of them are ready to offer data, one is selected non-deterministically. Thus, the sink node acts as a nondeterministic merger.

A complex connector has a graphical representation, called *Reo circuit or network*. The graph representing a connector is not directed. However, for each channel end x_c of a channel c, we use the directionality of x_c to assign a local direction on the neighborhood of $\hat{x_c}$ to the edge that represents c. Complex connectors are constructed out of simpler ones using the *join* operation. The join operation is defined only on nodes. Joining two nodes N_1 and N_2 destroys both nodes and produces a new node N, in which, $[N] = [N_1] \cup [N_2]$. This operation allows construction of arbitrary complex connector graphs involving any combination of channels picked from an open-ended set of channel types. The semantics of a connector is defined as a composition of the semantics of its constituent channels and nodes. Reo does not provide any channels, thus, it dose not define their semantics either. What Reo defines is the composition of channels into connectors and the semantics of this composition through the semantics of its three types of nodes.

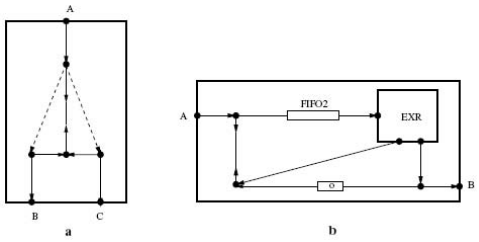


Figure 3.2: Exclusive router (a) and shift-lossy (b) channels designed by primitive channels of Reo [19]

As examples of Reo networks, Figure 3.2 shows the implementations of an exclusive router and a shift-lossy FIFO1 connectors in Reo. The intuitive behavior of the *exclusive router* is such that through its source node A, it obtains a data item d from its environment and delivers d to one of its sink nods B or C. If both B and C are ready to accept d, the exclusive router nondeterministically chooses one of them for delivery. A *shift-lossy FIFO1 channel* behaves the same as a FIFO1 channel, except that writing to its source end never blocks. If at the time of a write operation its buffer is full, the stored data item in the buffer is lost and the new data item replaces it in the buffer. For more examples and details of Reo circuits see [13, 14, 19, 30].

3.2 Basic Theory of Constraint Automata

Constraint automata were introduced as the semantics of Reo first in [30]. The semantics of constraint automata is based on timed data streams and their languages. In other words, constraint automata are defined as acceptors of the tuples of timed data streams and two constraint automata are (language - theoretically) equivalent if they accept exactly the same set of tuples of timed data streams.

In this section, we describe the basic theory of constraint automata, their semantics and their composition operators as introduced in [30].

3.2.1 Timed Data Streams

Before introducing the notion of constraint automata, we need to introduce some preliminary notations and the notions of timed data streams, their tuples and languages.

Definition 3.1 Let V be any set. We define the sets V^* and V^ω as the sets of all finite and infinite sequences (words or strings) over V, respectively. Obviously, we can define the set V^ω of all streams (infinite sequences) over V as the set of functions $w{:}\mathbb{N} \rightarrow V$. For a stream $w \in V^\omega$ we call $w(0)$ the initial value of w. The (stream) *derivative* w' of a stream w is defined as $w'(k) = w(k+1)$. We write $w^{(i)}$ for the i-th derivative of w which is defined by $w^{(0)} = w$ and $w^{(i+1)} = w^{(i)\prime}$. Note that $w^{(i)}(k) = w(i+k)$, for all $k, i \geq 0$.

Definition 3.2 Let $\mathcal{D}$ be a fixed and nonempty set of data that can be sent or received via channels. A *timed data stream* is an ordered pair $\langle \alpha, a \rangle$, in which, α is an infinite sequence of data and a is a time stream consisting of increasing positive real numbers that go to infinity. We denote the set of all timed data streams by TDS. Thus, more formally we have:

$$TDS = \{\langle \alpha, a \rangle \in \mathcal{D}^\omega \times \mathbb{R}_+^\omega |\ \forall n \geq 0\, (a(n) \langle a(n+1)\ and\ \lim_{n\to\infty} a(n) = \infty)\},$$

where $\mathbb{R}_+ = [0, \infty)$ is the set of all positive real numbers including zero.

A timed data stream $A = \langle \alpha, a \rangle$ represents occurrences of events at a port A and consists of a data stream $\alpha \in \mathcal{D}^\omega$ and a time stream $a \in \mathbb{R}_+^\omega$ of increasing positive real numbers. The time stream a indicates the moments $a(n)$ at which their respective data items $\alpha(n)$ occur at port A.

Let $\mathcal{N} = \{A_1, \ldots, A_n\}$ be a countable set of port names. With each port $A_i \in \mathcal{N}$, we associate a timed data stream recording both the data communicated and the time when the communication happens. That is, we define $TDS^{\mathcal{N}}$ as the set of all TDS-tuples consisting of one timed data stream for each port in $\mathcal{N}$.

Definition 3.3 Let $\mathcal{N} = \{A_1, \ldots, A_n\}$ be a countable set of port names. Then,

$$TDS^{\mathcal{N}} = \{(\langle \alpha_1, a_1 \rangle, \ldots, \langle \alpha_n, a_n \rangle) \mid \langle \alpha_i, a_i \rangle \in TDS,\ i = 1, \ldots, n\}$$

contains all *TDS-tuples* consisting of one timed data stream for each port.
A *TDS-language* L is a subset of $TDS^{\mathcal{N}}$, namely $L \subseteq TDS^{\mathcal{N}}$.

We use a family-notation $\theta = (\theta|_i)_{A_i \in \mathcal{N}}$ for the elements of $TDS^{\mathcal{N}}$, where $\theta|_i$ stands for the projection of θ along the port A_i. Simultaneous exchange of data between a set of ports can be detected by inspecting the time when communications happen. For this purpose, for $\theta \in TDS^{\mathcal{N}}$ we define $\theta.time$ to be a stream in $\mathbb{R}_+^\omega$ obtained by merging the streams $(\theta|_n)_r$ in increasing order. More formally,

Definition 3.4 Let $\theta = (\langle \alpha_1, a_1 \rangle, \ldots, \langle \alpha_n, a_n \rangle) \in TDS^{\mathcal{N}}$ be a TDS-tuple. $\theta.time$ is the time stream which, can be derived by merging the time streams $a_1, \ldots, a_n$ in an increasing order. Thus, for $\theta.time$ we have:

$$\theta.time(0) = min\{a_i(0) \mid i = 1, \ldots, n\},$$

and for all $j \geq 1$:

$$\theta.time(j) = min\{a_i(k) \mid a_i(k) > \theta.time(j-1),\ i = 1, \ldots, n,\ \ k = 0, 1, \ldots\}.$$

Also, $\theta.N$ is a *name-sets stream* over $2^{\mathcal{N}}$, in which, $\theta.N = \theta.N(0), \theta.N(1), \ldots$ and

$$\theta.N(k) = \{A_i \in \mathcal{N} \mid a_i(l) = \theta.time(k)\ for\ some\ l \in \{0, 1, 2, \ldots\}\}.$$

Let $\delta{:}N \to \mathcal{D}$ be a data assignment function, where, $N \neq \emptyset$ and $N \subseteq \mathcal{N}$. The notation of $\delta = [A \mapsto \delta_A : A \in N]$ designates the data assignment function that assigns to any TDS name $A \in N$ the value $\delta_A \in \mathcal{D}$. Now, $\theta.\delta = \theta.\delta(0), \theta.\delta(1), \ldots$, is a *data-assignments stream*, in which, $\theta.\delta(k)$ represents the observed data flow at time point $\theta.time(k)$, namely,

$$\theta.\delta(k) = [A_i \mapsto \alpha_i(l_i) : A_i \in \theta.N(k)]$$

where $l_i \in \{0, 1, 2, \ldots\}$ is the unique index with $a_i(l_i) = \theta.time(k)$.

In the rest of this section, we assume that the data set $\mathcal{D}$ is fixed and predefined. Thus, we do not mention it in the definitions.

3.2.2 Constraint Automata: the Operational Semantics of Reo

Constraint automata can be viewed as acceptors for tuples of timed data streams that are observed at certain ports $A_1, ..., A_n$. The rough idea is that such an automaton observes the data occurring at $A_1, ..., A_n$ and either changes its state according to the observed data or rejects the data if there is no corresponding transition in the automaton. Further, constraint automata are augmented with the names of their ports $A_1, ..., A_n$, where A_i stands for the ith TDS. Each transition in a constraint automaton is labeled with a pair N, g such that N is a non-empty subset of $\mathcal{N} = \{A_1, ..., A_n\}$, and g is a guard which, constrains data in the TDS of ports referenced in N.

Definition 3.5 Let $\mathcal{D}$ be a set of data and $\mathcal{N}$ be a set of port names. A *data constraint* g over sets $\mathcal{D}$ and $\mathcal{N}$ is a proposition that can be constructed using the following abstract grammar:

$$g ::= \mathit{true} \mid d_A = d \mid g_1 \vee g_2 \mid \neg g \qquad d \in \mathcal{D}, \quad A \in \mathcal{N}.$$

In the above grammar, for every port name $A \in \mathcal{N}$, d_A is a variable whose value at each time instance is the value of the data item exchanged through the port A at that time. Thus, $d_A = d$ means that the value of data on port A is d.

Let DC be the set of all data constraints over the names set $\mathcal{N}$ and the data set $\mathcal{D}$, defined by the above grammar. Obviously, DC contains some logically equivalent propositions. We use $DC(\mathcal{N}, \mathcal{D})$ as the set of all logically different data constraints over the names set $\mathcal{N}$ and the data set $\mathcal{D}$. In other words, $DC(\mathcal{N}, \mathcal{D})$ is the partitioned version of DC using the logical equivalence relation. Thus, for all data constraints equivalent to g, we use the notion of $g \in DC(\mathcal{N}, \mathcal{D})$. Obviously, if $\mathcal{N}$ and $\mathcal{D}$ are finite then $DC(\mathcal{N}, \mathcal{D})$ is finite.

Now, we introduce the notion of constraint automaton as originally has been defined in [30]:

Definition 3.6 A *constraint automaton* is a quadruple $C = \langle Q, \mathcal{N}, \longrightarrow, Q_0 \rangle$ where,
Q is a set of states,
$\mathcal{N}$ is a set of names,
$\longrightarrow \subseteq Q \times 2^{\mathcal{N}} \times DC \times Q$ is a set of transitions,
$Q_0 \subseteq Q$ is the set of initial states.
We write $p \xrightarrow{N,g} q$ instead of $(p, N, g, q) \in \longrightarrow$ and call N the name set and g the guard of the transition. For every transition $p \xrightarrow{N,g} q$ it is supposed that $N \neq \emptyset$ and $g \in DC(N, \mathcal{D})$.

A constraint automaton $C = \langle Q, \mathcal{N}, \longrightarrow, Q_0 \rangle$ is finite, if the sets $Q, \mathcal{N}$ and $\mathcal{D}$ are finite. Also, C is said to be *deterministic* if its set of initial states Q_0 is a singleton and for each state q, a set of port names N and a data assignment $\delta{:}N \rightarrow \mathcal{D}$ there is at most one transition $q \xrightarrow{N,g} q'$ with $\delta \models g$.

The intuitive operational behavior of a constraint automaton is as follows. It starts in its initial state q_0. If the current state is q, then C waits until data items occur at some of its ports

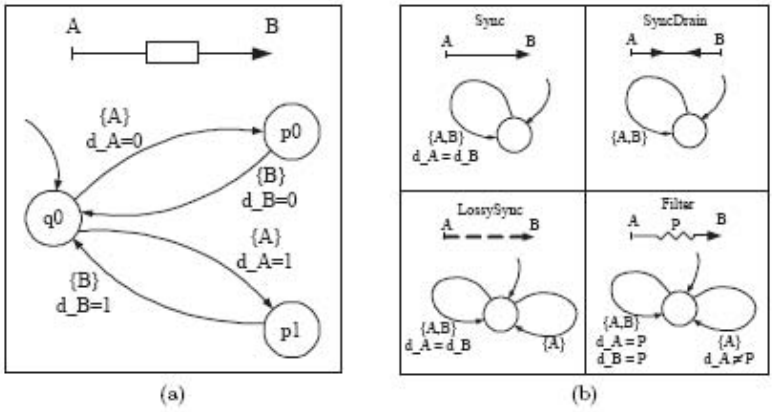


Figure 3.3: Constraint automata for some basic channels in Reo [30]

$A_1, ..., A_n$. Suppose data item d_1 occurs at A_1 and data item d_2 at A_2 while (at this moment) no data is observed at the other ports $A_3, ..., A_n$. This triggers the automaton to check the data constraints of the outgoing transitions of state q with a name set $\{A_1, A_2\}$ to choose a transition t, such that its guard is satisfied by $A_1 \mapsto d_1$ and $A_2 \mapsto d_2$ resulting in state p. If there is no $\{A_1, A_2\}$-transition from q whose data constraint is fulfilled then C remains waiting (perhaps indefinitely).

Example 3.1 Figure 3.3 shows the constraint automata models of the set of Reo channels which we introduced in the previous section [30]. Figure 3.3(a) is the constraint automaton model of a FIFO1 channel from a source node A to sink B over the data set $\{0, 1\}$. Figure 3.3(b) shows the constraint automata models of the remaining channels with the consideration that $d_A = d_B$ is an abbreviation for $\vee_{d \in \mathcal{D}}(d_A = d \wedge d_B = d)$ where $\mathcal{D}$ is the set of data $\{0, 1\}$. In the case of the filter channel it must be that $P \subseteq \mathcal{D}$.

Like ordinary automata are acceptors of finite strings, constraint automata are acceptors of tuples of timed data streams. Informally, each element of the tuple is associated with a port of the system and corresponds to the streams of observed data communicated through this port together with the time when the data has been observed.

Definition 3.7 Let $C = \langle Q, \mathcal{N}, \longrightarrow, Q_0 \rangle$ be a constraint automaton and φ be a TDS-tuple, $\varphi \in TDS^{\mathcal{N}}$.
- An *infinite run* for φ in C is an infinite sequence of states $r = q_0, q_1, ...,$ in which, $\forall i, q_i \in Q$ and there exists transition $q_0 \xrightarrow{N,g} q_1$ with $N = \varphi.N(0)$, $\varphi.\delta(0) \models g$ and r' is an infinite run for φ'. The infinite run $r = q_0, q_1, ...$ is an *initial infinite run*, if $q_0 \in Q_0$.
- TDS-tuple φ is accepted by a constraint automaton C if and only if there is an initial infinite run for φ in C. The language of constraint automaton C is

$$L_{TDS}(C) = \{\varphi \in TDS^{\mathcal{N}} \mid C \text{ accepts } \varphi\}.$$

The above definition of $L_{TDS}(C)$ can also be formally defined by means of the greatest fixed point of a suitably chosen monotone operator [30]. For the purpose of this thesis we found it easier to reason with the accepted language characterized by means of the (standard) notion of accepted runs.

As for the case of finite automata, using Rabin-Scott powerset construction for each nondeterministic constraint automaton there is a deterministic one which accepts the same language [30].

Further, in a finite constraint automaton C all transitions with unsatisfiable guards can be removed without any effect on the TDS language accepted by C, where a guard g of a transition $p \xrightarrow{N,g} q$ is said to be semantically unsatisfiable for N if there is no data assignment for elements of N which satisfies g (take, for example, g to be $\neg true$). In the rest of this thesis we assume without any loss of generality that all guards in a constraint automaton are satisfiable with respect to the set of names of the transition they belong to.

Remark 3.1 As we mentioned in Definition 3.6, in [30] it is presupposed that for every transition $p \xrightarrow{N,g} q$ the set of ports N is non-empty. This condition makes constraint automata the operational models of the *observable* behavior of systems' components. In other words, with this condition constraint automata are the operational models of the *interfaces* of components. If we ignore this condition, a constraint automaton may also have transitions of the form $p \xrightarrow{\emptyset,true} q$. We abbreviate this form of transitions by $p \xrightarrow{\tau} q$ and call them by τ-transitions. We refer to constraint automata that are allowed to have τ-transitions by the term *constraint automata with τ-transitions*. τ-transitions can be considered as models of internal behaviors of the components that are not exactly observable in the interfaces.

Remark 3.2 Constraint automata are, in general, not closed under complement. Informally this is due to the fact that constraint automata do not have *final* states. If we augment the definition of constraint automaton by a set of final states and use Büchi acceptance condition (a timed data stream is accepted if at least one of the correspondent runs for it contains one of the final states infinitely many times), we refer to the resulting automaton as a *Büchi constraint automaton*. Obviously, a constraint automaton is a Büchi constraint automaton in which all states are accepting. Its complement, however does not need to satisfy this property.

3.2.3 Composing of Constraint Automata

In the literature on constraint automata, there are two operators for composing them: join of two constraint automata and hiding a name in all transitions of a constraint automaton [30].

Join

Constraint automata can be composed by means of a join operator, the semantic counterpart of the join operator in Reo [30]. Different than the ordinary product for finite automata, the composition of two constraint automata is allowed even if they have different alphabets. In fact, the resulting constraint automaton has transitions when data occur at the ports belonging to only one of the automata, without involving the transitions or states that it inherits from the other automaton (because at that point in time, there is no data on any of its corresponding ports). More formally, the join operation for constraint automata is defined as follows:

Definition 3.8 Let $C_1 = \langle Q_1, \mathcal{N}_1, \longrightarrow_1, Q_{01}\rangle$ and $C_2 = \langle Q_2, \mathcal{N}_2, \longrightarrow_2, Q_{02}\rangle$ be two constraint automata (with or without τ-transitions). The *join* of C_1 and C_2 is the constraint automaton:

$$C_1 \bowtie_C C_2 = \langle Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, \longrightarrow, Q_{01} \times Q_{02}\rangle$$

where, transition relation $\longrightarrow$ is defined by the following rules:
Rule 1:

$$\frac{q \xrightarrow{N_1, g_1}_1 q', p \xrightarrow{N_2, g_2}_2 p', N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1}{\langle q, p\rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle q', p'\rangle}$$

Rule 2:

$$\frac{q \xrightarrow{N_1, g_1}_1 q', N_1 \cap \mathcal{N}_2 = \emptyset}{\langle q, p\rangle \xrightarrow{N_1, g_1} \langle q', p\rangle},$$

Rule 3 (dual of rule 2):

$$\frac{p \xrightarrow{N_2, g_2}_2 p', N_2 \cap \mathcal{N}_1 = \emptyset}{\langle q, p\rangle \xrightarrow{N_2, g_2} \langle q, p'\rangle}.$$

Basically, the two automata have to agree on the data exchanged on the common ports (that is, the names used in the transition of the first automaton known to the second automaton are exactly the same as the names used by the transition of the second automaton known to the first one), and each maintains its own behavior on the other ports (as described by the last two rules).

The join of two constraint automata using the operation defined in Definition 3.8 is correct with respect to the join of their accepted TDS-languages, where the join of two TDS-languages is basically the same as defined in the theory of relational databases [30]: the projection on the common indexes (port names) of the resulting language must agree with that of the two original languages, while the projection on each indexes in one language but not in the other must agree with the projection on the same index of the language where the index belongs to. Thus, it can be shown that [30]:

Lemma 3.1 Let $C_1 = \langle Q_1, \mathcal{N}_1, \longrightarrow_1, Q_{01}\rangle$ and $C_2 = \langle Q_2, \mathcal{N}_2, \longrightarrow_2, Q_{02}\rangle$ be two constraint automata. Then:

1) $L_{TDS}(C_1 \bowtie_C C_2) = L_{TDS}(C_1) \bowtie L_{TDS}(C_2)$,

2) If $\mathcal{N}_1 = \mathcal{N}_2$ then $L_{TDS}(C_1 \bowtie_C C_2) = L_{TDS}(C_1) \cap L_{TDS}(C_2)$,

where for TDS-languages L_1 and L_2, $L_1 \bowtie L_2$ means the standard join of the languages of tuples in the theory of databases.

Example 3.2 Let us join the constraint automata representing two FIFO1 channels, one from source A to sink B and the other from source B to sink C. The automata models of the two channels and the resulting automaton are illustrated in Figure 3.4. For simplicity we assume that the data set is $\mathcal{D} = \{d\}$. Thus, every transition label contains only the set of port names that participate in firing the transition.

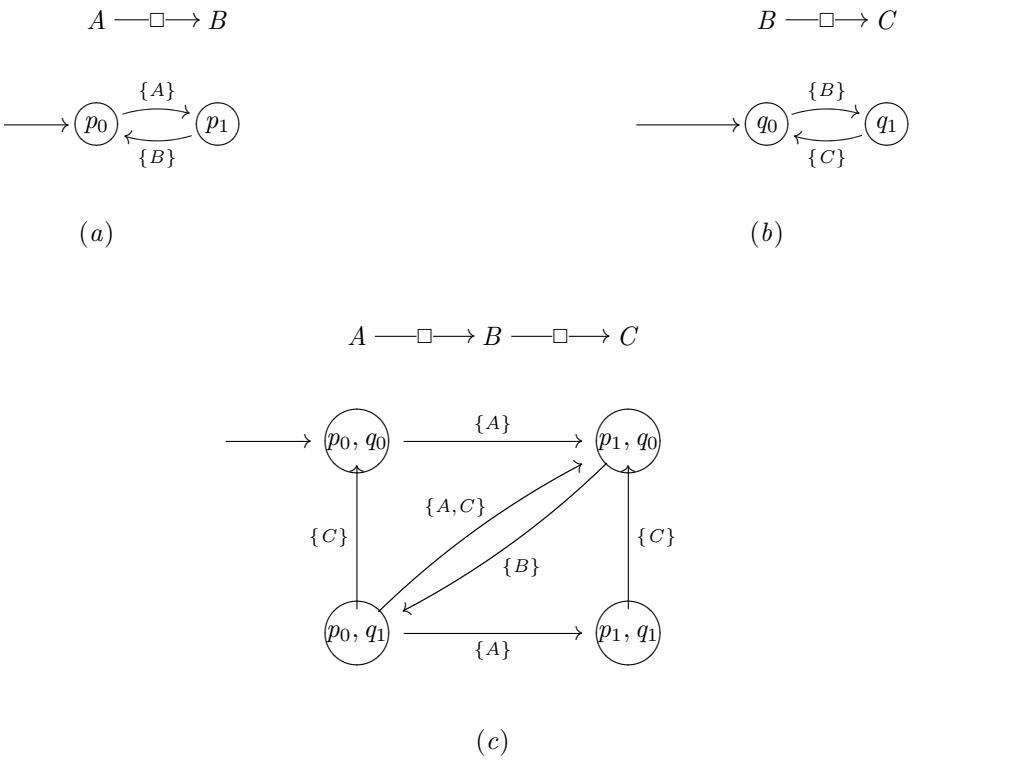


Figure 3.4: Joining of constraint automata models of two FIFO1 channels

Hiding of port Names

Now, we define hiding of a port name in all transitions of a constraint automaton. First consider constraint automata without τ-transitions. The hiding operation is defined as follows [30]:

Definition 3.9 Let $C = \langle Q, \mathcal{N}, T, Q_0 \rangle$ be a constraint automaton and $B \in \mathcal{N}$. The constraint automaton resulted by *hiding* of B in C is constraint automaton

$$\exists B[C] = \langle Q, \mathcal{N} \backslash \{B\}, \longrightarrow_B, Q_{0,B} \rangle$$

where transition relation $\longrightarrow_B$ is defined as follows:
Let $\longrightarrow^*$ be the transition relation such that $q \longrightarrow^* p$ if and only if there exists a finite path $q \overset{\{B\},g_1}{\longrightarrow} q_1 \overset{\{B\},g_2}{\longrightarrow} q_2 \ldots \overset{\{B\},g_n}{\longrightarrow} q_n$, in which, $q_n = p$ and $g_1, g_2, \ldots, g_n$ are satisfiable. Then,

$$Q_{0,B} = Q_0 \cup \{p \in Q | q_0 \rightarrow^* p, \textit{ for some } q_0 \in Q_0\}.$$

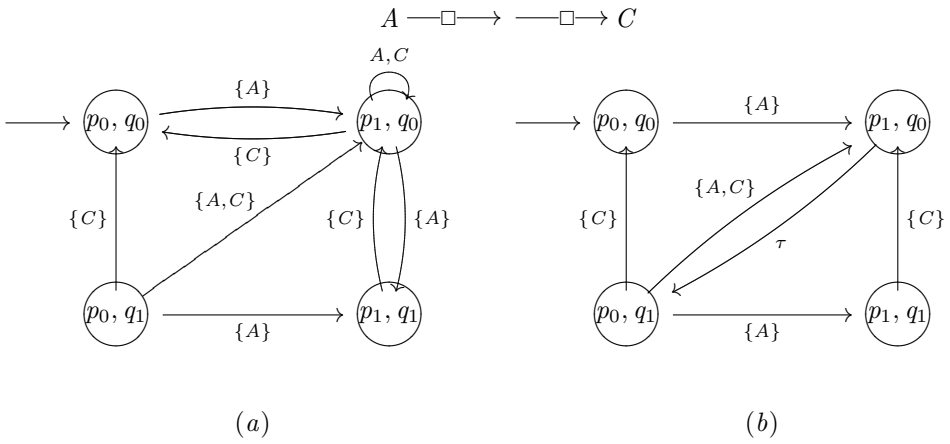


Figure 3.5: Hiding of port B in constraint automaton of Figure 3.4(c)

The transition relation $\rightarrow_B$ is defined by the rule:

$$\frac{q \rightarrow^* p, p \xrightarrow{N,g} r, N' = N \backslash \{B\} \neq \emptyset, g' = \exists B[g]}{q \xrightarrow{N',g'}_B r}$$

where $\exists B[g] = \vee_{d \in \mathcal{D}} g[d_B/d]$.

Example 3.3 Let us hide the intermediate port B in the constraint automaton illustrated in Figure 3.4(c) using Definition 3.9. The result is illustrated in Figure 3.5(a). As we expect the resulting automaton is the operational model of the observable behaviors of a FIFO channel with capacity of two. Also, observe that in this case the state $\langle p_0, q_1 \rangle$ is unreachable from the initial state, and can thus be removed.

Now, let us consider the more general case of constraint automata with τ-transitions. In this case, the definition of hiding is simpler, because it is allowed for the hiding operation to generate τ-transitions.

Definition 3.10 Let $C = \langle Q, \mathcal{N}, T, Q_0 \rangle$ be a constraint automaton (with or without τ-transitions) and $B \in \mathcal{N}$. The constraint automaton (possibly with τ-transitions) resulting from *hiding* of B in C is the constraint automaton

$$\exists B[C] = \langle Q, \mathcal{N} \backslash \{B\}, \longrightarrow_B, Q_{0,B} \rangle$$

where the transition relation $\longrightarrow_B$ is defined as follows:

$$\frac{q \xrightarrow{N,g} p, N' = N \backslash \{B\}, g' = \exists B[g]}{q \xrightarrow{N',g'}_B p}$$

and $\exists B[g] = \vee_{d \in \mathcal{D}} g[d_B/d]$.

Example 3.4 Let us hide the intermediate port B in the constraint automaton illustrated in Figure 3.4.c using Definition 3.10. The result is illustrated in Figure 3.5.b. As we expect, the resulting automaton is the operational model of a FIFO channel with capacity of two, considering its internal actions.

Remark 3.3 Obviously, if we hide a port name of a constraint automaton using Definition 3.9, the resulting automaton is language theoretically equivalent with the automaton that is the result of hiding the same port using Definition 3.10 and then eliminating all τ-transitions using the standard algorithm of eliminating all τ-transitions (ϵ-transitions) of ordinary finite automata (for this algorithm see [66]). For example, the constraint automaton illustrated in Figure 3.5.a is the result of eliminating the τ-transition in the constraint automaton illustrated in Figure 3.5.b.

3.3 Other Semantic Models for Reo

In recent years, several models have been proposed in the literature to formally capture the semantics of Reo connectors. The first formal operational model for Reo was constraint automaton that we introduced in the previous section. In the next chapters, we will introduce our proposed operational semantics for Reo using standard notion of Büchi automata. Before introducing our proposed model, in this section, we briefly review the other important semantic models of Reo and discusse some of their shortcomings in the context of the questions and aims of this thesis.

3.3.1 Co-algebraic Model of Connectors

The *co-inductive calculus of timed data streams* first proposed in [27] and then extended to the notion of *Abstract Behavior Types (ABT in short)* [14], is a simple and transparent relational model of Reo. In this calculus the behavior of a connector port is modeled as a timed data stream. As we showed in Definition 3.2, an element of the time-stream is the time at which its respective data value in the data-stream is observed at its corresponding port. The interactions of connectors are modeled as relations on timed data streams. More formally, let TDS be the set all timed data streams, then each basic connector, namely channel, is defined as a binary relation

$$R \subseteq TDS \times TDS$$

over timed data streams. For example, the synchronous channel of Reo (Sync) is modeled by the following relation:

$$Sync = \{(\langle \alpha, a \rangle, \langle \beta, b \rangle) \,|\, \alpha = \beta \wedge a = b\}$$

where $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$ respectively are the timed data streams over input and output ports of the channel.

Because the connectors are relations, their composition is modeled by relational composition. For example, the composition of two copies of the synchronous channel yields the

following binary relation over the set TDS:

$$Sync \circ Sync = \{(\langle \alpha, a\rangle, \langle \beta, b\rangle) \mid \exists \langle \gamma, c\rangle : (\alpha = \gamma \wedge a = c) \wedge (\gamma = \beta \wedge c = b)\}$$

This co-inductive model abstracts away from the connector topology, and the direction of the dataflow within the connector. Connectors are reduced to a collection of ports, and the behavior of the component-based system is expressed as a relation. Based on the relational nature of this semantics of Reo, if someone be interested to verify some properties of the communication protocol modeled by composed connectors, it should be done using a deductive verification method, as for a simple example it has been done in [27].

These models of Reo were shown to be equivalent to constraint automata, and thus unable to express several fairness constraints and context dependencies [36, 37]. Also, because the model is not an operational model it is not suitable for model checking based verification.

3.3.2 Connector Coloring Models

Connector coloring is an intuitive semantics for Reo to model dataflow behavior of connectors [47]. Colors are used to denote the presence of dataflow and its absence in connected ports. Colorings with two colors suffice to express the same class of behavior as constraint automata can express [52]. Each coloring of a connector is a solution to the synchronization constraints imposed by its channels and nodes.

By refining the set of colors to three colors and propagating the negative information about the absence of dataflow in some ports, a set of context dependencies can be expressed.

Coloring a connector in a specific state with given boundary conditions (I/O requests) provides a means to determine the routing alternatives for dataflow. The circuit representation of connectors are used to describe the dataflow behavior: each coloring corresponding to a dataflow behavior of the connector can be overlaid on top of the circuit representation to provide insight into the dataflow behavior of the individual primitives of the circuit [52]. The product composition operator for colorings has been defined such that it is associative, commutative, and idempotent. These properties make the coloring scheme with its composition operator suitable for distributed implementations [52]. In a recent work, Jongmans investigates the relationships between 2/3-color coloring models and constraint automata through a set of operators that transform one model to the other [84]. In another paper, he establishes an encoding of context sensitivity expressed in 3-coloring semantics within constraint automata [83].

Coloring based semantics is not suitable for the purpose of verification, especially by model checking which is the main purpose of this thesis. This type of semantics for Reo and its extension called tile logic [21] suffer from a number of other problems. For a survey of these problems see [36].

3.3.3 Intentional Automata

The main reason for introducing *intentional automata* [52] as an operational semantic model of Reo was to overcome the shortcoming of constraint automata in expressing context dependent behaviors of connectors. An intentional automaton explicitly models the arrival of

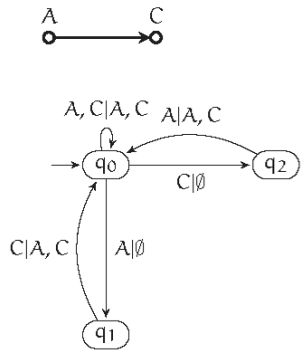


Figure 3.6: Intentional automaton model of a synchronous channel [52].

communication (or I/O) requests, and distinguishes between communication requests and the actual communications. This gives the model an extra degree of expressiveness that becomes useful in modeling systems that need to behave differently depending on the presence or absence of pending requests in their context/environment. Intentional automata are transition systems over the set of port names where the context dependencies are expressed by labeling transitions with a request set and a firing set. The request set models the context and the firing set models the subsequent behavior. More formally, in an intentional automaton over a set of ports $\mathcal{N}$, each transition from state q into state q' is of the form $q \xrightarrow{R \mid F} q'$, where $R \subseteq \mathcal{N}$ is the set of ports requesting data communication and $F \subseteq \mathcal{N}$ is the set of ports that actually participate in data communication during this transition. Also, the pending (arrived but not fired) communication requests are modeled by expanding the set of states.

For example, the intentional automaton model of a synchronous channel from input port A to output port C with the ability to suspend data communication when one of the ports is not ready to communicate, as modeled in [52], is illustrated in Figure 3.6. In this model, q_1 is the state in which port A is requesting communication while C is not ready, and similarly, q_2 is the state in which port C is requesting communication while A is not ready. The transition $q_0 \xrightarrow{A,C \mid A,C} q_0$ means that whenever both ports A and C are requesting communication, the communication is fired on them simultaneously.

Intentional automata can be composed using a product and a hiding operators very similar to their counterparts for constraint automata (see Definitions 3.8 and 3.10). The only defined semantics (equivalence relations) for intentional automata are weak and strong bisimulation relations.

Because of the strategy of modeling pending request by expanding the set of states, intentional automata have quiet a large number of states to manage the buffering and firing of such requests, which rapidly become difficult to manipulate, making intentional automata not suitable for model checking purposes [36]. While intentional automaton tries to overcome the shortcomings of constraint automaton for context dependent behaviors of components, based on the absence of the notion of final (accepting) states or sets of fairness constraints in its

syntax, it fails to express several fairness requirements.

3.3.4 Guarded and Reo Automata

Following the basic idea of intentional automaton (namely expressing request and firing sets explicitly in each transition) and similar to our guarded strings semantics for augmented Büchi automata of records (will be introduced in Chapter 5), Bonsangue *et al.* introduced a new model for Reo connectors. In general, this model is called *guarded automaton* and when it is used with some constraint to model context dependent connectors, it is called *Reo automaton* [36, 37].

Guarded automata are transition systems over a set of port names (an alphabet set in a more general view) where the context dependencies are expressed by labeling transitions with a boolean guard expression over the alphabet as their atoms and a firing set. The guard expresses the context and the firing set models the subsequent behavior. More formally, in a guarded automaton over a set of ports $\mathcal{N}$, each transition from a state q into a state q' is of the form $q \xrightarrow{g \mid f} q'$, where g is a boolean expression over the set of ports (as the atoms) and $f \subseteq \mathcal{N}$ is the set of ports that actually participate in data communication during this transition. Guarded automata are defined as acceptors of finite guarded strings, where each guarded string is a string of alternating sets of atoms of the boolean algebra of the guard expressions and sets of port names. Two guarded automata are equivalent if their languages of guarded strings are the same. In addition to this equivalence, a notion of bi-simulation relation between guarded automata has been defined such that it also implies the language equivalence. Furthermore, a product operation as the counterpart of the intersection of the languages of guarded automata has been defined.

Reo automata are guarded automata such that each transition label $g \mid f$ satisfies two criteria: *reactivity* which guaranties that data flow only on ports where I/O requests are made; and *uniformity* which captures two properties: first, that the condition of the request set corresponding precisely to the firing set is sufficient to cause firing, and second, that removing additional unfired requests from a transition will not affect the behavior of the connector [37]. It has been shown that the set of Reo automata is closed under the product of guarded automata [37].

In comparison with an intentional automaton, a Reo automaton is more dense, namely, to model context dependencies and suspended communications, it does not expand the set of state as much as an intentional automaton does. Also the product of Reo automata is more difficult to compute than the product of intentional automata and the information represented in the states of intentional automata is represented in Reo automata using compound transition labels. It can be shown that the expressive power of Reo automata is at least as much as that intentional automata. Both intentional and Reo automata have been proposed to express more semantics of Reo connectors without significant attempts to use them in verification and model checking. Because of their complicated syntax and non-standard semantics (from standard automata theory point of view) defining temporal logics over them and designing model checking algorithms will not be so simple. Also, similar to intentional automaton and based on the absence of the notion of final (accepting) states or sets of fairness constraints in the syntax of Reo automaton, it fails to express several fairness requirements.

3.3.5 Process Algebraic and Structural Operational Semantics

In addition to the above mentioned models, there are some other attempts to present formal semantics for component connectors. In the work of Barbosa et al. [32] the semantics of components are defined by the expressions of a process algebra. These expressions carry out both positive (presence) and negative (absence) information about data in ports. Complex connectors are built from basic ones using some composition operators such as join, parallel composition, and interleaving. This work needs more investigation to before it can be used for the purpose of deductive verification and is not suitable for model checking. The model needs more enhancements to express some important fairness constraints.

In the joint works of Kokash, Krause et al., reported in [91, 92, 93, 94], and [95], they map some semantic models for Reo, namely, constraint automata, timed constraint automata, and coloring semantics to the process algebraic specification language of mCRL2 [61] and show the correctness of their mappings. Also, they have build a tool for these mappings that is now a part of the tool-set *Extensible (Eclipse) Coordination Tools* (ECT) [2].

In another work, Mousavi et al. [116] express the formal semantics of Reo connectors by a set of structural operational semantic rules. In general, this semantics is not a context dependent semantics, it carries only positive information. However, to express some context dependencies, specially for the lossy synchronous channel, the authors used extra more rules to remove undesired behavior. To be able to express more fairness or context dependent behavior the set of rules needs to be expanded.

3.4 Tool Support for Reo

In addition to a mostly updated homepage for Reo [5] that helps its user to have access to the research and tools supporting Reo, there is a tool-set, called *Extensible (Eclipse) Coordination Tools* (ECT), for Reo [2]. It is a set of integrated plug-ins for the *Eclipse platform*, which offer a graphical development environment for the specification, analysis and execution of component-based software systems using the coordination language Reo [95]. It contains a set of tools implemented by a variety of researchers and programmers. Based on the last report of the tool-set ECT, presented in [95], it now contains the following tools:

- *Graphical Reo editor* by which the user can graphically construct Reo networks out of the basic channel types.
- *Animator tool* that generates animations for the flow of data in Reo nets.
- *mCRL2 conversion tool* which encodes Reo specifications generated by the graphical Reo editor to the mCRL2 specification language. This conversion is based on the works reported in detail in [91, 92, 93, 94], and [95].
- *Extensible automata editor*. Because there are several automata based models for Reo, ECT contains a tool for deriving automata based models from Reo in general. The framework contains a graphical automata editor and can also be used outside of the context of Reo [95].

- Tools for *stochastic modeling and analysis*. There are two tools for stochastic modeling and analysis in ECT: one that generates a sort of intentional automata augmented by stochastic information, called *quantitative intensional automata*, from the graphical Reo models [113, 114, 22, 23, 26], and the other which simulates the behavior of Reo nets [150, 87] using their coloring semantics as described in [47].

- *Execution engines* that execute Reo connectors. Currently there exist two implemented execution engines: one that generates the codes based on constraint automata models and the other that executes Reo connectors in a distributed platform (for more see [125, 95]).

- *Conversion tools* that convert some other modeling languages such as UML2 and BPMN to Reo.

- *Vereofy*. As we mentioned before, *Vereofy* is a model checker implemented based on the work of Baier et al. reported in [99, 98]. Briefly, it does symbolic model checking of a CTL-like temporal logic (called *BTSL* temporal logic) by using ordered binary decision diagrams (OBDD) as the data structures representing constraint automata models. The *Vereofy* tool is available through its own web-page [7] and now is also accessible through the ECT tool-set [2].

4 Fair Component Connectors

In this chapter, we introduce Büchi automata of records and unconditional fair constraint automata as alternative models for the operational semantics of Reo. We compare the expressiveness of these models with that of the original model of constraint automata discussed in the previous chapter. In the first section, we review some shortcomings of constraint automata and of their TDS based semantics in modeling component connectors and motivate the use of records and Büchi automata of records as operational semantics for Reo. In the second section, we introduce the notions of records, streams and languages of records. We also give a bidirectional translation of TDS-languages and record-based languages. The notion of Büchi automaton of records (BAR) is introduced in the third section and we show that every constraint automaton can be translated into a Büchi automaton of records. In the fourth section, we show that BAR's can be used to model Reo connectors especially connectors with some fairness conditions on their behavior using some examples. Therefore, BARs are semantically more powerful than constraint automata. In the fifth section, a set of composition operators for BARs is introduced. We compare the join operator of BARs with its counterpart for constraint automata and show that the join operation can be obtained using two more basic operations, namely, product and alphabet extension. In the subsequent section, we introduce the notion of unconditional fair constraint automata and compare their expressiveness with that of constraint automata and Büchi automata of records. In the last section, we introduce a version of constraint automaton, called *fair constraint automaton*, whose syntax is the same as constraint automaton except that now it has final (accepting) states, but its semantics is based on the languages of streams of records.

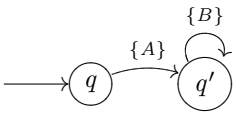
4.1 Introduction

In the previous chapter, we have seen that constraint automata are operational models of Reo connectors. However, we can recognize some shortcomings in using constraint automata as the semantics for Reo. These shortcomings can be categorized in two main groups, those related to the TDS-based semantics of constraint automata and those about the modeling capabilities for Reo connectors. Briefly, the first group shows that the TDS-based semantics of constraint automata is more concrete than what is needed in modeling Reo connectors, whereas the other group shows that constraint automata fail to model all expected behaviors of the connectors. The main shortcomings concerning the TDS-language based semantics of constraint automata can be summarized as follows.

1- Constraint automata are defined as the acceptors of timed data streams. However, timed data streams are much more concrete than constraint automata, because they record the actual times when communications happen, whereas constraint automata record just the temporal order of data communications (and not their times).

2- Different than finite and Büchi automata, the simplicity of a constraint automaton is not necessarily reflected by the TDS language it recognizes:

Example 4.1 Consider for example the following constraint automaton on two ports A and B over a singleton data set:



While the automaton describes only a *single* event happening at port A, a TDS-tuple θ accepted by the automaton consists of a pair of two *infinite* sequence of events θ_A and θ_B, one describing the data-flow at port A and the other the flow at port B, such that all events in θ_B happen between the first and the second event in θ_A. All events but the first in θ_A are really irrelevant, yet one needs to describe them all.

In addition, constraint automata fail to model some expected behaviors of Reo connectors. For instance, they cannot model *fairness constraints* over the behaviors of a connector, as well as operations that depend on pending I/O operations on the communication ports of a connector. This latter feature is called *context dependency*, which occurs when the behavior of a connector can change depending upon not only the presence of requests on a connector boundary, but also on their absence. In such cases, the behavior of a connector can change dramatically with changing context. In this chapter, we concentrate on the issue of fairness constraints. In the next chapter we will discuss context dependencies.

Fairness Constraints Many specification formalisms for reactive and concurrent systems incorporate some notions of fairness constraints such as *unconditional*, *weak*, and *strong fairness*. Informally, the requirement of unconditional fairness disallows executions of the system in which certain sets of actions or situations are taken only finitely many times. In other words, in a model with an unconditional fairness constraint we are interested only in the executions wherein a certain set of actions (or a certain set of the states of the system) are seen infinitely many times. The requirements of weak and strong fairness are conditional. The weak fairness requirement disallows executions in which certain sets of actions or situations are continually enabled but not taken. Namely, the weak fairness requirement states that continually enabled actions or states must occur infinitely often. The requirement of strong fairness disallows executions in which certain sets of actions or states are enabled infinitely often but they are taken only finitely many times. That is, if certain actions or situations are enabled infinitely often then they must occur infinitely often. More formally:

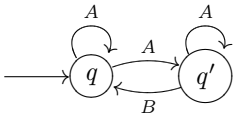
Definition 4.1 A *basic transition system* is a tuple $A = \langle Q, \Sigma, \Delta, q_0 \rangle$ where, Q is a finite set of *states*, Σ is a finite nonempty set of symbols called *alphabet*, $\Delta \subseteq (Q \times \Sigma \times Q)$ is a *transition relation*, and $q_0 \in Q$ is the *initial* state. An infinite *run* of A is an infinite sequence $\rho = q_0, a_0, q_1, a_1, ...$, of alternating states and symbols where, for all i, $(q_i, a_i, q_{i+1}) \in \Delta$. We say that the symbol (action) $a \in \Sigma$ is *enabled* in state $q \in Q$ whenever there exists a transition $(q, a, q') \in \Delta$. Also, we say that the symbol a is taken (or occurs) in position $i \in \mathbb{N}$ of the infinite run $\rho = q_0, a_0, q_1, a_1, ...$ if $a_i = a$. Let ρ be an infinite run in the basic transition system A and $F \subseteq \Sigma$ be a set of symbols. Then,

- ρ is *unconditionally F-fair* if an element in F occurs infinitely often in ρ.
- ρ is *strongly F-fair* if the condition that infinitely often an element in F (not necessarily the same) is enabled implies that an element in F (not necessarily the same) occurs infinitely often in ρ.

- ρ is *weakly F-fair* if the condition that some element in F (not necessarily the same) is eventually always enabled implies that an element in F (not necessarily the same) occurs infinitely often in ρ.

Next we present some simple examples of unconditional fairness constraint in the context of component connectors.

Example 4.2 Consider a channel from port A to port B with a buffer with capacity of one. Suppose that if the buffer is empty, the input data from port A can be saved in the buffer or can get lost. If the buffer is full all other inputs are lost. When the buffer is full, port B is able to get the saved data and then the buffer becomes empty. We call this channel a *Restive-Buffer* channel and model it with the following basic transition system:

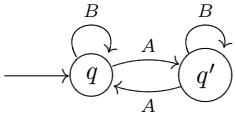


Now, an example of an unconditional fairness constraint is given by considering the (infinite) runs in which the buffer becomes full infinitely many times. Namely, the fair runs are the executions in which state q' (or transition with label B) is taken infinitely many times. In these runs, it is impossible that all input data get lost.

Obviously, if we consider the above transition system as a finite automaton over infinite words (a Büchi automaton) with state q' as final state, the semantics of the model is exactly what we want.

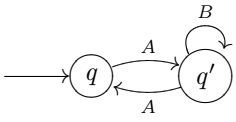
In general, it can be shown that unconditional fairness conditions correspond to the Büchi acceptance condition in the theory of automata on infinite words [101]. Based on this fact, sometimes unconditional fairness conditions are called as Büchi fairness conditions [131].

Example 4.3 Now consider the following basic transition system:



Suppose that we want the model to be weakly fair with respect to the action set $\{B\}$. A transition with action B is continuously enabled in all infinite runs of the above model. A run is weakly fair if it takes transitions with action B infinitely many times. Thus, the run $q, A, q', A, q, A, q', A, \ldots$ and every run in which there are only finitely many transitions with action B are (weakly) unfair with respect to action set $\{B\}$.

Example 4.4 Take the following basic transition system:



Suppose that we want the model to be strongly fair with respect to action set $\{B\}$. In all infinite runs of the above model the transition with action B is enabled infinitely often. An run is strongly fair if it takes the transition with action B infinitely many times. Thus, run $q, A, q', A, q, A, q', A, \ldots$ and every runs in which there are only finitely many transitions with action B are (strongly) unfair with respect to action set $\{B\}$.

In general, strong fairness conditions correspond to the Streett acceptance condition in the theory of automata on infinite words [101]. Interestingly, Streett automata can be efficiently simulated by Büchi automata [130]. Thus, a semantics for component connectors based on Büchi automata is able to specify both unconditional and strong fairness constraints. In this chapter, our main goal is to present this kind of semantics for Reo connectors.

Fair connectors. According to the view point of exogenous coordination, a connector (coordinator) is an open system. By the term of *open system*, we mean that the set of actions (in the case of automata models, the set of transition labels or symbols) are not under the control of the connector. They are fired by the environment. Thus, it makes sense to talk about fairness for the behavior of the system only when there is non-determinism. For deterministic systems/automata, one cannot really talk about their fairness. Whenever a system/automaton has non-deterministic choices it becomes meaningful to expect it to behave fairly. There are different definitions for non-determinism for specification formalisms. To fix our terminology, we call a Reo connector a *non-deterministic connector* if there are possible alternative firings of the ports that the connector can decide to choose (and if the connector *can* decide to make a transition, then staying in the current state is *not* a choice). If a connector is non-deterministic, we can augment its specification or model by fairness constraints.

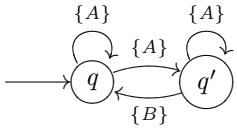
On the other hand, a transition system or automaton can be considered as the model of a *closed system*. For instance, a connector and its environment together can be considered as a closed system and modeled by an automaton. In this case, we can distinguish fair and unfair runs of the system by augmenting its model by fairness constraints.

Thus, in our terminology, speaking about the fair connectors is permitted only for connectors that have non-deterministic choices in their behavior. Also, if an automaton is considered as the model of a connector or an open system and there is non-determinism in its behavior, it can be asked to be fair. However, if we speak generally about fairness for automata models, they should be considered as models of closed systems.

Next, we show that constraint automata are not always able to model the desired fairness conditions, even in the simplest case, namely the unconditional fairness.

Constraint Automata and Fairness. The timed data streams semantics of constraint automata implicitly expresses some unconditional fairness constraints. Let us have an example:

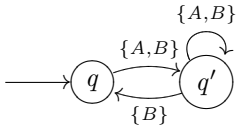
Example 4.5 We model the Restive-Buffer channel that we presented in Example 4.2 with the following constraint automaton:



According to the TDS-languages semantics of constraint automata, in the above automaton port A cannot be fired eventually always because TDS-language semantics forces to assign an infinite sequence of time-data pairs to both ports A and B. Thus, using the TDS based semantics of constraint automata implicitly satisfy the unconditional fairness constraint of Example 4.2.

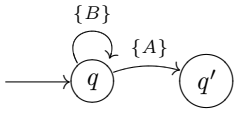
However, there are several cases that the TDS-languages based semantics of constraint automata fails to satisfy some simple fairness conditions:

Example 4.6 Consider the following constraint automaton:



The automaton accepts TDS-languages in which B alone never occurs (even though it is always enabled in state q'). Note that if we consider the above automaton as a Büchi automaton with two simple action names $\{A, B\}$ and $\{B\}$ and two accepting state then action $\{B\}$ can occur alone.

Now, consider the following constraint automaton:



The automaton does not accept any timed data streams tuple, because A cannot appear only once (even if B is initially enabled in state q).

The above example in addition to Example 4.1 show that the TDS-language based semantics of constraint automata sometimes is not able to express fairness constraints but sometimes implicitly it does! Furthermore, timed data streams contain exact time value expressions while in data passage through ports only the temporal orderings of data exchanges are of interest. Thus, TDS-language semantics of constraint automata is more concrete than it is necessary[1].

In this chapter, we introduce the notion of Büchi automaton of records as the alternative operational semantics for Reo with a more standard and simpler semantics which is able to express the desired unconditional and strong fairness conditions. We use records as data

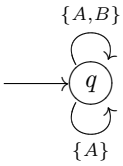
[1] In a later work and in order to define the operational semantics of timed Reo connectors, some of the authors of constraint automata, introduced the notion of *scheduled data streams* [18, 17]. This formalism is similar to our proposed *streams of records* from the view point that both abstract away the exact time stamps and focus only on the ordering.

structures for modeling the simultaneous executions of events: ports in the domain of the record are allowed to communicate simultaneously the data assigned to them, while ports not in the domain of the record are blocked so that they can not participate in communication. The behavior of a network of components is given in terms of (infinite) sequences of records, so to specify the order of occurrence of the events. Standard operational models can be used to recognize such languages. For example, we use ordinary Büchi automata as operational devices for recognizing languages of streams of records. Because our model is based on Büchi automata, we can easily express *fairness conditions* admitting only executions for which some actions occur infinitely many times [145]. In the next chapter, we enrich the model to overcome the context dependency problems.

For example, for the lossy synchronous channel which has been introduced in the previous chapter, we will define at least two types of *fair lossy synchronous* channels:

Example 4.7 A lossy synchronous channel from port A to port B behaves as a synchronous channel except that the input data can be non-deterministically lost or delivered to the sink. We call this channel as a *ND-LossySync* channel. If we add the fairness condition that not all data can get lost, we call the channel as a *weak fair LossySync* channel. If we consider stronger fairness condition that only finitely many data can get lost, the channel is called as a *strong fair LossySync* channel.

In [30] a lossy synchronous channel is modeled using the following (deterministic) constraint automaton:



For the moment, suppose that the above model is a basic transition system with infinite traces semantics (or it is a Büchi automaton whose only state is accepting). From the environments viewpoint, firing each one of the two actions can be selected non-deterministically. Thus, it is possible that the transition with action $\{A\}$ is selected forever or the other transition to be selected only finitely many time. In the first case, the model violates both weak and strong fairness conditions. In the other, the model violates the strong fairness constraint. But, what about the above model if it is regarded as a constraint automaton with the TDS-language semantics? As we explained before, the TDS-language semantics forces to assign an infinite sequence of time-data pairs to both ports A and B. Thus, it implicitly satisfies the fairness constraint that not all data at port A get lost.

We also show that every constraint automaton with a *slight correction* of their TDS semantics can be translated into an essentially equivalent Büchi automaton of records. The construction of the Büchi automaton is straightforward and the result may appear as not surprising at all. But beware! The languages recognized by the two type of automata have different structures. In fact it is easy to embed a language on streams of records into a language of timed data streams, but not vice-versa. Despite these structural differences, we show that the converse also holds without losing any information as far as constraint automata is

concerned. An immediate consequence of this result is that, since Büchi automata enjoy closure properties that constraint automata do not have, our model is more expressive. In fact we give a few concrete examples of realistic connectors (not considered in the Reo language until now) that can be specified in our model but not with constraint automata.

The main reason for having time information in the timed data streams is compositionality with respect to the Reo join operator. We introduce a join composition operator for Büchi automata on streams of records and show that it is correct with respect to the join operator for constraint automata. Also, we present a method to recast this join operation using the standard product operator of Büchi automata.

4.2 Streams and Languages of Records

Now we introduce records as data structures for modeling the simultaneous executions of events: ports in the domain of the record are allowed to communicate simultaneously the data assigned to them, while ports not in the domain of the record are blocked from participating communication. The behavior of a network of components is given in terms of (infinite) sequences of records, so to specify the order of occurrence of the events.

Definition 4.2 Let $\mathcal{N}$ be a finite nonempty set of (port) names and $\mathcal{D}$ a finite nonempty set of data.
(1) We write $Rec_{\mathcal{N}}(\mathcal{D}) = \mathcal{N} \rightharpoonup \mathcal{D}$ for the set of *records* with entries from the set of data $\mathcal{D}$ and labels from the set of names $\mathcal{N}$, consisting of all partial functions from $\mathcal{N}$ to $\mathcal{D}$.
(2) For a record $r \in Rec_{\mathcal{N}}(\mathcal{D})$ we write $dom(r)$ for the domain of r.
(3) Sometimes we use the more explicit notation $r \doteq [n_1 = d_1, \ldots, n_k = d_k]$ for a record $r \in Rec_{\mathcal{N}}(\mathcal{D})$, with $dom(r) = \{n_1, \ldots, n_k\}$ and $r(n_i) = d_i$ for $1 \leq i \leq k$. Different than a tuple, the order of the components of a record is irrelevant and its size is not fixed a priori.
(4) We denote by τ the special record with the empty domain, that is $dom(\tau) = \emptyset$.
(5) A *stream of records* over a data set $\mathcal{D}$ and a name set $\mathcal{N}$ is an infinite string of records $w \in Rec_{\mathcal{N}}(\mathcal{D})^{\omega}$.
(6) A *language of (streams of) records* over a data set $\mathcal{D}$ and a name set $\mathcal{N}$ is a set of infinite strings of records $L \subseteq Rec_{\mathcal{N}}(\mathcal{D})^{\omega}$.

We use records as data structures for modeling constrained synchronization of ports in $\mathcal{N}$. Following [127], we see a record $r \in Rec_{\mathcal{N}}(\mathcal{D})$ as carrying both positive and negative information: only the ports in the domain of r have the possibility to exchange the data assigned to them by r, while the other ports in $\mathcal{N} \setminus dom(r)$ are definitely constrained to *not* perform any communication. This intuition is formalized by the fact that only for ports $n \in dom(r)$ data can be retrieved, using *record selection* $r.n$. Formally, $r.n$ is just a (partial) function application $r(n)$.

Further, positive information may increase by means of the *update* (and extension) operation $r[n: = d]$, defined as the record with the domain $dom(r) \cup \{n\}$ mapping the port n to d and remaining invariant with respect to all other ports. The *hiding* operator '$\setminus$' is used to increase the negative information. For $n \in \mathcal{N}$, the record $r \setminus n$ hides the port n to the environment by setting $dom(r \setminus n) = dom(r) \setminus \{n\}$, and $(r \setminus n).m = r.m$.

Definition 4.3 Let $r_1 \in Rec_{\mathcal{N}_1}(\mathcal{D})$ and $r_2 \in Rec_{\mathcal{N}_2}(\mathcal{D})$.
(1) We say that records r_1 and r_2 are *compatible*, if $dom(r_1) \cap \mathcal{N}_2 = dom(r_2) \cap \mathcal{N}_1$ and for all $n \in dom(r_1) \cap dom(r_2)$, $r_1.n = r_2.n$.
(2) The *union* of compatible records r_1 and r_2, denoted by $r_1 \cup r_2$, is a record over port names $\mathcal{N}_1 \cup \mathcal{N}_2$, such that, for all $n \in dom(r_1)$, $(r_1 \cup r_2).n = r_1.n$ and for all $n \in dom(r_2)$, $(r_1 \cup r_2).n = r_2.n$.

4.2.1 Bidirectional Translation of Record and TDS-Languages

Let us compare the expressiveness of TDS-languages with that of languages of streams of records. First, we introduce a slight modification in the definition of timed data stream:

Definition 4.4 Let $\mathcal{N}$ be a fixed finite set of *port names* and $\mathcal{D}$ a non-empty set of *data* that can be communicated through those ports. The set TDS of all (infinite) *timed data streams* over $\mathcal{D}$ consists of all pairs $\langle \alpha, a \rangle \in \mathcal{D}^\omega \times \mathbb{R}_+^\omega$ such that

1. for all $k \geq 0$ either $a(k) = \infty$ or $a(k) < a(k+1)$, and
2. $\lim_{k \to \infty} a(k) = \infty$.

where $\mathbb{R}_+ = [0, \infty]$ is the set of all positive real numbers including zero and infinity.

The only difference of the above definition of timed data stream and the original one (see Definition 3.2) is that in the present definition the time value ∞ (infinity) is also allowed. This simplifies our next discussions and will solve some of the problems [2].

For instance in the case of Example 4.1 it is enough to consider the values of all events time but the first in θ_A to be infinity. The definitions of TDS-tuples and TDS-languages remain the same as previously defined.

Given a TDS-language L for $\mathcal{N}$ we can abstract from its timing information to obtain a set of streams over $Rec_{\mathcal{N}}(\mathcal{D})$. For a TDS-tuple $\theta \in TDS^{\mathcal{N}}$, the idea is to construct a stream of records $\Upsilon(\theta) \in Rec_{\mathcal{N}}(\mathcal{D})^\omega$, where, for each k, the record $\Upsilon(\theta)(k)$ contains all ports and data exchanged at time $\theta.time(k)$. In fact, we define for each $n \in \theta.N(k)$ and $k \in \mathbb{N}$,

$$\Upsilon(\theta)(k).n = \theta.\delta(k)_n$$

Note that $dom(\Upsilon(\theta)(k)) = \theta.N(k)$. As usual, we extend this construction to sets, namely, for every $L_{TDS} \subseteq TDS^{\mathcal{N}}$,

$$\Upsilon(L_{TDS}) = \bigcup \{\Upsilon(\theta) \mid \theta \in L_{TDS}\}.$$

Example 4.8 Let

A	d	d'	d"	...
	0.5	0.7	1.9	
B	d	d'		...
	0.5	1.2		

[2] In [16], Arbab uses the $\perp$ symbol in a footnote as a special value for the time values in time streams to model *finite behavior*. This is similar to using ∞ as we have here.

be a TDS-tuple over port set $\{A, B\}$. Then,

$$\rho = [A = d, B = d][A = d'][B = d'][A = d"] \cdots$$

is its correspondent stream of records. The time stamps are used only to determine the ordering of data communications.

Conversely, any stream of records $\rho \in Rec_{\mathcal{N}}(\mathcal{D})^{\omega}$ generates a TDS-language $\Theta(\rho)$ by *guessing* the times when data are exchanged so to respect the relative order of communication imposed by ρ. Formally,

$$\Theta(\rho) = \{\theta \mid \forall k \geq 0{:}(\theta.N(k) = dom(\rho(k)) \wedge \forall n \in dom(\rho(k)){:}\theta.\delta(k)_n = \rho(k).n)\}\,.$$

Example 4.9 For example, for ρ being the stream of records as in Example 4.8 above, the following TDS-tuple

A	d	d'	d"	⋯
	1	10.4	23.6	
B	d	d'		⋯
	1	10.5		

is in the language $\Theta(\rho)$. Clearly, also the TDS-tuple in Example 4.8 is an element of the same language.

We extend Θ to languages $L \subseteq Rec_{\mathcal{N}}(\mathcal{D})^{\omega}$ by setting

$$\Theta(L) = \bigcup\{\Theta(\rho) \mid \rho \in L\}.$$

The function $\Theta{:}2^{Rec_{\mathcal{N}}(\mathcal{D})^{\omega}} \rightarrow 2^{TDS^{\mathcal{N}}}$ is an embedding of languages over records into TDS-languages for $\mathcal{N}$.

Lemma 4.1 For each $L \subseteq Rec_{\mathcal{N}}(\mathcal{D})^{\omega}$, $L = \Upsilon(\Theta(L))$.

Proof.
Let $\rho \in L$ be a stream of records. Since $\rho = \Upsilon(\Theta(\rho))$ we have $L \subseteq \Upsilon(\Theta(L))$.
Now Let $\rho \in \Upsilon(\Theta(L))$. There are a stream of records $\rho' \in L$ and a TDS-tuple $\theta \in TDS^{\mathcal{N}}$ such that $\rho = \Upsilon(\theta)$ and $\theta = \Theta(\rho')$. Thus, θ is a proper time assignment into ρ' and ρ is the time abstraction of θ. Obviously it should be $\rho = \rho'$. Thus, $\Upsilon(\Theta(L)) \subseteq L$. □

The counterpart of the above lemma for TDS-languages does not hold, because a tuple of time data stream $\theta \in TDS^{\mathcal{N}}$ may contain specific time information that gets lost when mapped into a stream of record $\Upsilon(\theta)$. In the next section we see that for constraint automata the information lost in the above translation is never used.

4.3 Büchi Automata of Records

Sets of streams of records are just languages of infinite strings, and as such some of them can be recognized by ordinary Büchi automata. Next, we recall some basic definitions and facts on Büchi automata [138].

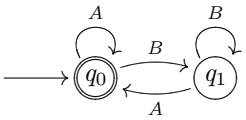


Figure 4.1: A Büchi automaton for L in Example 4.10

4.3.1 Büchi Automata: A Review

A *Büchi automaton* is a non-deterministic finite state automaton which takes infinite words as input. A word is accepted if the automaton goes through some designated *final* or *accepting* state infinitely often while reading the word. More formally:

Definition 4.5
(1) A *Büchi automaton* is a tuple $B = \langle Q, \Sigma, \Delta, Q_0, F\rangle$ where, Q is a finite set of *states*, Σ is a finite nonempty set of symbols called *alphabet*, $\Delta \subseteq (Q \times \Sigma \times Q)$ is a *transition relation*, $Q_0 \subseteq Q$ is a nonempty set of *initial* states and $F \subseteq Q$ is a set of *accepting (final)* states.
(2) An *infinite computation* for a stream $\omega = a_0, a_1, \cdots \in \Sigma^\omega$ in B is an infinite sequence $q_0, a_0, q_1, a_1, ...$, of alternating states and alphabet symbols in which $q_0 \in Q_0$ and $(q_i, a_i, q_{i+1}) \in \Delta$ for all i.
(3) The language accepted by a Büchi automaton B consists of all streams $\omega \in \Sigma^\omega$ such that there is an infinite computation for ω in B with at least one of the final states occurring infinitely often. The language of a Büchi automaton B, denoted by $L(B)$, is the set of all streams accepted by it.
(4) We say that two Büchi automata B_1 and B_2 are *(language-based) equivalent* if $L(B_1) = L(B_2)$.
(5) Let $B = \langle Q, \Sigma, \Delta, Q_0, F\rangle$ be a Büchi automaton. B is called as a *deterministic* Büchi automaton if $|Q_0| \leq 1$ and the transition relation Δ can be considered as a function of the form $\Delta{:}(Q \times \Sigma) \to Q$.

If we regard the state space of a Büchi automaton as a graph, an accepting computation (or run) traces an infinite path which start at some state $q_0 \in Q_0$, reaches an accepting state $q_F \in F$ and, thereafter, keeps looping back to q_F infinitely often. In graphical representation of Büchi automata accepting states are distinguished from other states by drawing them with a double circle.

Example 4.10 Consider the alphabet $\Sigma = \{A, B\}$. Let $L \subseteq \Sigma^\omega$ consist of all infinite words α such that there are infinitely many occurrences of A in α. Figure 4.1 shows a Büchi automaton recognizing L. The initial state is marked by an arrow without a source. There is only one accepting state q_0 which is indicated by a double circle. In this automaton, all transitions labeled A lead into the accepting state and, conversely, all transitions coming into the accepting state are labeled A. From this, it follows that the automaton accepts an infinite word if and only if it has infinitely many occurrences of A.

The complement of L, which we denote $\bar{L}$, is the set of all infinite words α such that α has only finitely many occurrences of A. An automaton recognizing $\bar{L}$ is shown in Figure 4.2.

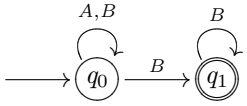


Figure 4.2: A Büchi automaton for $\bar{L}$ in Example 4.10

The automaton guesses a point in the input beyond which it will see no more A's - such a point must exist in any input with only a finite number of A's. Once it has made this guess, it can process only B's - there is no transition labeled A from the second state, so if it reads any more A's it gets stuck.

In the above example, notice that the automaton recognizing L is deterministic while the automaton for $\bar{L}$ is non-deterministic. It can be shown that the non-determinism in the second case is unavoidable - that is, there is *no* deterministic automaton recognizing $\bar{L}$. This means that Büchi automata are fundamentally different than their counterparts on finite inputs: we know that over finite words, deterministic automata are as powerful as non-deterministic automata. In other words, non-deterministic Büchi automata are strictly more powerful than deterministic Büchi automata: there are languages recognized by non-deterministic Büchi automata that cannot be recognized by any deterministic Büchi automaton [138].

Generalized Büchi Automata In several applications, other types of automata on infinite objects are useful. In fact, there are several variants of automata on infinite words that are equally expressive as nondeterministic Büchi automata, although they use more general acceptance conditions. For some of these automata, the deterministic version has the full power of nondeterministic Büchi automata. Muller, Rabin and Streett automata are examples of these types of automata on infinite words [138]. Also, Büchi automaton itself has some slight variants, called *generalized* and *alternating Büchi automata*, both of which are equally expressive as nondeterministic Büchi automata. For the purpose of this thesis, it suffices to consider *generalized (nondeterministic) Büchi automata*. The difference between a Büchi automaton and a generalized Büchi automaton is that the acceptance condition of the generalized one requires to visit several sets (of final states) $F_1, \ldots, F_k$ infinitely often. More formally:

Definition 4.6
(1) A *generalized Büchi automaton* is a Büchi automaton $B = \langle Q, \Sigma, \Delta, Q_0, \mathcal{F} \rangle$ but for the set of final states, that now is a set of sets, that is, $\mathcal{F} \subseteq 2^Q$.
(2) A stream $\omega \in \Sigma^\omega$ is accepted by generalized Büchi automaton B if and only if there is an infinite computation π for ω in B such that for every $F \in \mathcal{F}$ at least one of the states in F occurs in π infinitely often.

The definitions of languages recognized by generalized Büchi automata and their equivalence are the same as for the case of Büchi automata.

Example 4.11 Figure 4.3 shows a generalized Büchi automaton over the alphabet set $\Sigma = \{A, B, C\}$ with the acceptance sets $F_1 = \{q_1\}$ and $F_2 = \{q_2\}$. The accepted language

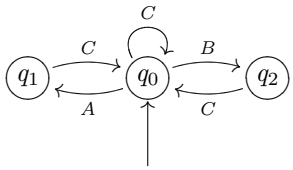


Figure 4.3: A generalized Büchi automaton with the set of accepting sets $\mathcal{F} = \{\{q_1\}, \{q_2\}\}$.

consists of all infinite words over the alphabet set $\Sigma = \{A, B, C\}$ such that both A and B hold infinitely often (possibly at different positions).

Remark 4.1 The set $\mathcal{F}$ of accepting sets of a generalized Büchi automaton may be empty. If $\mathcal{F} = \emptyset$ the stream ω is accepted if and only if there exists an infinite computation for ω in the automaton. Note the difference with the case of an ordinary Büchi automaton whose set of final states is empty. For a Büchi automaton whose set of final states is empty, there are *no* accepting computations and the language of the automaton is empty. Contrary to that, *every* infinite computation of a generalized Büchi automaton with $\mathcal{F} = \emptyset$ is accepting.

Clearly, every Büchi automaton is a generalized Büchi automaton with a singleton set of final states, containing the original set of of final states. Conversely, every generalized Büchi automaton can be transformed into an equivalent Büchi automaton:

Lemma 4.2 [3] Let $B = \langle Q, \Sigma, \Delta, Q_0, \mathcal{F} \rangle$ be a generalized Büchi automaton. Then, there exists a Büchi automaton B' such that $L(B) = L(B')$.

Proof. Based on Remark 4.1, if $\mathcal{F} = \emptyset$ then B accepts all infinite strings over the alphabet Σ. In this case, B is equivalent with the Büchi automaton that has only one state, say q, that is both initial and final, and for every $a \in \sigma$, there is a self-transition (q, a, q) in B.

Now, we assume that $\mathcal{F} \neq \emptyset$. Let $\mathcal{F} = \{F_0, \ldots, F_{k-1}\}$, where $k \geq 0$. The basic idea of the construction of B' is to create k copies of B such that the accepting set F_i of the ith copy is connected to the corresponding states of the $i + 1$th copy. The acceptance condition for B' consists of the requirement that an accepting state of the first copy is visited infinitely often. This ensures that all other accepting sets F_i of the k copies are visited infinitely often too.

Now we can define ordinary Büchi automaton $B' = \langle Q', \Sigma, \Delta', Q'_0, F' \rangle$ such that:

- $Q' = Q \times \{0, \ldots, k-1\}$,
- $Q'_0 = Q_0 \times \{0\}$,
- $F' = F \times \{0\}$.

The transition relation $\Delta' \subseteq (Q' \times \Sigma \times Q')$ is defined as follows. For all $q \in Q$, $A \in \Sigma$, and $i \in [0..k-1]$:

[3] This lemma is a known result in the literature of Büchi automata. In the rest of this chapter we will use the construction procedure that is introduced in its proof.

- if $q \notin F_i$, then for all $q' \in Q$ that $(q, A, q') \in \Delta$, $(\langle q, i\rangle, A, \langle q', i\rangle) \in \Delta'$,
- else, for all $q' \in Q$ that $(q, A, q') \in \Delta$, $(\langle q, i\rangle, A, \langle q', (i+1)\ mod\ k\rangle) \in \Delta'$.

Now, we can simply show that $L(B) = L(B')$ (for more detail of the proof see for example [29]). □

4.3.2 Büchi Automata on Streams of Records

In the rest of this chapter, we work with Büchi automata whose alphabet sets are defined as sets of records over some sets of port names and data:

Definition 4.7 Let $\mathcal{N}$ be a finite set of port names and $\mathcal{D}$ a finite set of data. Also, let $B = \langle Q, \Sigma, \Delta, Q_0, F\rangle$ be a Büchi automaton over the alphabet $\Sigma = Rec_{\mathcal{N}}(\mathcal{D})$. We call B as a *Büchi automaton (on streams) of records*, abbreviated by BAR.

In the following example we show that the basic channels of Reo can be modeled by BARs. Thus, not only it will be an example for BARs, but also this example shows the expressive power of BARs as the semantic model of component connectors.

Example 4.12 In Figure 4.4 we show BAR models of the basic Reo channels. We assume that all channels are from port A to port B and the data set is $\mathcal{D} = \{d, d'\}$. Sometimes instead of drawing separate loops on the same vertex, we draw one loop with several labels separated by commas. For the case of filter we assume that the *filter value* is d. The non-deterministic lossy synchronous channel (ND-LossySync) that we model in Figure 4.4.d is the same as we introduced in Example 4.7. As we mentioned in that example, using fairness assumptions, at least two other versions of the lossy synchronous channel can be defined. Later in this chapter, we will show that using BARs, we are also able to model these two fair lossy synchronous channels (see Section 4.4).

Also, it can be shown that the source and sink nodes in the Reo terminology should be modeled as special kinds of connectors. A source node acts as a *duplicator* channel while a sink node acts as a *merger* (for more detail see Chapter 3). For simplicity of our discussions, in the following example we use duplicator and merger connectors to explicitly show their behavior as Reo primitive and show that they can be modeled by BARs.

Example 4.13 A *duplicator* is a connector with a source and two sink ends. Whenever an entity at the source is ready to put data and the entities at both sinks are ready to get it, data will be delivered from the source to the sinks of this connector synchronously. Thus, a duplicator can be modeled as we have shown in Figure 4.5. Again we assume that the data set is $\mathcal{D} = \{d, d'\}$.

Now, consider the *merger* connector with two source ports A and B and one sink port B. Intuitively, it transmits synchronously data item from either A or B to the port C. If both the source ports A and B offer data at the same time then only one of them is chosen non-deterministically. The Büchi automaton of records model of this connector, when the data set is $\mathcal{D} = \{d, d'\}$, is shown in Figure 4.6.

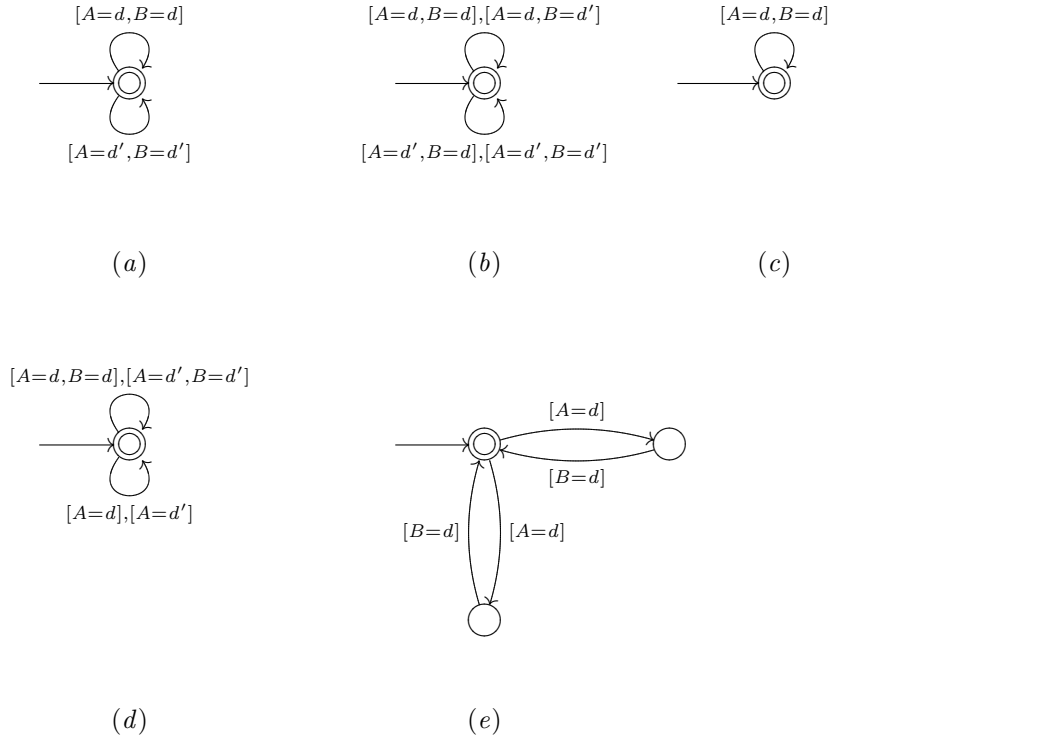


Figure 4.4: BAR models of basic Reo channels: a) Sync channel b) SyncDrain channel, c) Filter channel, (d) ND-LossySync channel, and (e) FIFO1 channel.

In general, a Büchi automaton of records may contain transitions labeled by τ. These can be considered as internal actions, as no port of the system can be involved in a communication. Since they are externally invisible we may ignore them. However, if we remove all τ symbols from a stream of records ω, the resulting sequence need not to be infinite anymore. For example, removing all τ's from the stream consisting of only τ symbols will result in the empty (and hence finite) string.

Definition 4.8 Let B be a Büchi automaton of records. The *visible language* of B is defined as:

$$L_{vis}(B) = \{\rho \in Rec_{\mathcal{N}}(\mathcal{D})^{\omega}, \mid \exists \omega \in L(B) : \rho = vis(\omega)\},$$

where $vis(\omega)$ denotes the sequence obtained by removing all τ symbols from ω. We say that automata B_1 and B_2 are *visibly equivalent* if $L_{vis}(B_1) = L_{vis}(B_2)$.

Note that $L_{vis}(B)$ contains only infinite sequences and therefore is a subset of the set of sequences obtained from removing the τ's from the streams in $L(B)$. For example, if $L(B) = \{[A = d] \cdot [A' = d'] \cdot \tau^{\omega}\}$, then $L_{vis}(B) = \emptyset$, because removing all τ's from a stream

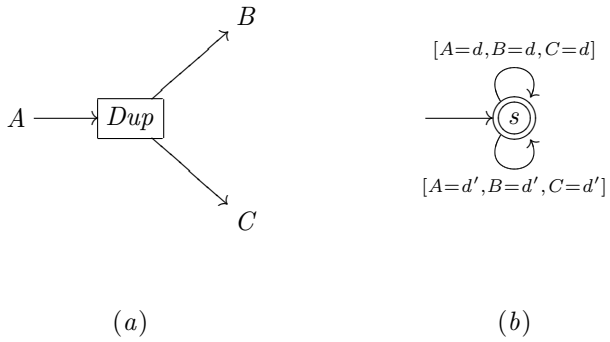


Figure 4.5: A duplicator channel and its BAR model

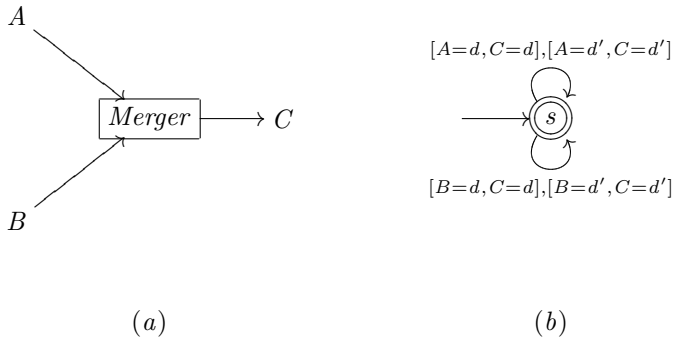


Figure 4.6: An (unfair) merger channel and its BAR model

consisting of infinitely many τ's will result in a finite string, and thus not in $Rec_{\mathcal{N}}(\mathcal{D})^{\omega}$. Clearly, $L_{vis}(B) = L(B)$ if B does not have τ-transitions.

Example 4.14 In Figure 4.7 two visibly equivalent BAR models are illustrated. To simplify the figure, we use a singleton data set $\mathcal{D} = \{d\}$ and denote a record labeling a transition only by the domain where it is defined.

By a simple generalization of the standard algorithm for eliminating the ϵ-transitions of an ordinary finite automaton over finite words [66], we can construct a Büchi automaton recognizing $L_{vis}(B)$.

Lemma 4.3 For every Büchi automaton of records B there is a Büchi automaton of records B' (without τ-transition) such that, $L_{vis}(B) = L(B')$.

Proof. Let $B = \langle Q, \Sigma, \Delta, Q_0, F\rangle$ be the Büchi automaton of records over the alphabet $\Sigma = Rec_{\mathcal{N}}(\mathcal{D})$. Using B, we construct the following BAR without τ-transitions:

$$B' = \langle Q', \Sigma', \Delta', Q_0', F'\rangle$$

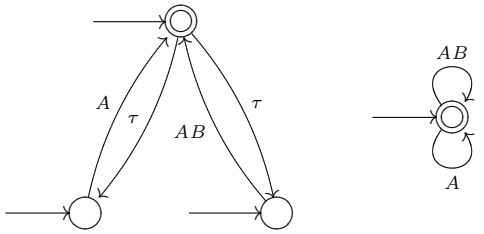


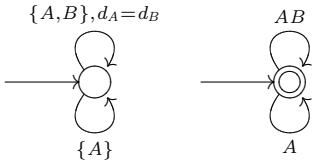
Figure 4.7: Two visibly equivalent Büchi automata of records.

such that,

- $Q = Q'$,
- $\Sigma' = \Sigma - \{\tau\}$,
- $Q_0 = Q'_0$,
- $F' = F \cup \{q | \exists q_F \in F, (q \xrightarrow{\tau^+} q_F) \in \Delta\}$,
- $(q, a, q') \in \Delta' \iff (q \xrightarrow{\tau^* a \tau^*} q') \in \Delta$

where, by $(q \xrightarrow{\tau^+} q_F) \in \Delta$ we mean that using the transition relation Δ there is a finite path π from q to q_F such that $\pi = q \xrightarrow{\tau} q_F$ or for $k \geq 1$, $\pi = q \xrightarrow{\tau} q_1 \xrightarrow{\tau} \ldots q_k \xrightarrow{\tau} q_F$; and by $(q \xrightarrow{\tau^* a \tau^*} q') \in \Delta$ we mean that there is a finite path π from q to q' such that there are states q_1 and q_2 (not necessarily distinct from each other or from q and q') where, $\pi = q \xrightarrow{\tau^*} q_1 \xrightarrow{a} q_2 \xrightarrow{\tau^*} q'$ in B.

Now, we can show that $L_{vis}(B) = L(B')$. Obviously, $L_{vis}(B) \subseteq (\Sigma')^\omega$. First, let $\rho \in L_{vis}(B)$, there is $\omega \in \Sigma^\omega$ in $L(B)$ such that $\rho = vis(\omega)$. Thus, there is an accepting infinite computation $\pi = q_0, a_0, q_1, a_1, \ldots$ such that at least one of the accepting states, say $q_F \in F$, occurs infinitely often (looping style) in π and $\omega = a_0 a_1 \ldots$. Consider π' as the computation obtained by replacing all finite subcomputations of the form $q_i \quad q_j$ in π with q_i. Because both ρ and ω are infinite words, by the definition of B', π' is an accepting computation for ρ in B'. Thus, $\rho \in L(B')$. Conversely, suppose that $\rho \in L(B')$. Thus, there is an accepting infinite computation $\pi' = q'_0, a_0, q'_1, a_1, \ldots$ in B'. Using the definition of δ', for every triple (q_i, a, q_j) in computation π' there is a computation fragment $q_i \xrightarrow{\tau^* a \tau^*} q_j$ in B. Replace all triples of the form (q_i, a, q_j) in computation π' with one of the corresponding computation fragments $q_i \xrightarrow{\tau^* a \tau^*} q_j$ and call the resulting computation π. Obviously, using the definitions of Δ' and F', it is necessary that π be an accepting infinite computation for an infinite word $\omega \in \Sigma^\omega$ such that $\rho = vis(\omega)$. Thus, $\omega \in L(B)$ and $\rho \in L_{vis}(B)$. □



(a) (b)

Figure 4.8: Models of a *non-deterministic* lossy synchronous channel by a) a constraint automaton and b) a Büchi automaton of records.

4.3.3 Recasting Constraint Automata into Büchi Automata

Now we show that for every constraint automaton A over a name set $\mathcal{N}$ and a data set $\mathcal{D}$ we can construct a Büchi automaton of records. The key observation is that for each transition labeled (N, g) in A, there is a set of (total) data assignments $\{\delta{:}N \to \mathcal{D} \mid \delta \models g\}$. Every data assignment in this set can be seen as a partial function from $\mathcal{N}$ to $\mathcal{D}$, with domain $N \subseteq \mathcal{N}$, that is, it is a record in $Rec_{\mathcal{N}}(\mathcal{D})$. We can thus construct a Büchi automaton of records $B(A)$ with the same (initial) states as A, with all states as final, and with transitions labeled by each of the above data assignment for every transition in A.

Definition 4.9 For every constraint automaton $A = \langle Q, \mathcal{N}, \longrightarrow, Q_0 \rangle$ over a finite data set $\mathcal{D}$ and a finite name set $\mathcal{N}$, we define $B(A)$ to be the Büchi automaton of records $\langle Q, Rec_{\mathcal{N}}(\mathcal{D}), \Delta, Q_0, F \rangle$, where $F = Q$ and Δ is the following set of transitions:

$$\{(q, r, q') \mid \exists q \xrightarrow{(N,g)} q', \exists \delta{:}N \to \mathcal{D}{:}\delta \models g, dom(r) = N \text{ and } \forall n \in N{:}r.n = \delta(n)\}.$$

Example 4.15 Consider the constraint automaton depicted in Figure 4.8(a). It models a non-deterministic lossy synchronous channel from the source A to the sink B: data in $\mathcal{D}$ either flow from A to B or they get lost after they are read by A [30]. Figure 4.8(b) shows the corresponding Büchi automaton on streams of records. Again, to simplify the figure, we use a singleton data set $\mathcal{D} = \{d\}$ and denote records only by the domains where they are defined.

All Büchi automata of records in Figure 4.4 are obtained as the translations of the constraint automata models of the same channels in Figure 3.3. Note that in Figure 4.4 the data set is $\mathcal{D} = \{d, d'\}$.

The following theorem shows that timed data streams are not different than streams of records, at least as far as finite constraint automata are concerned.

Theorem 4.4 Let $A = \langle Q, \mathcal{N}, \longrightarrow, Q_0 \rangle$ be a finite constraint automaton. Then,

$$\Upsilon(L_{TDS}(A)) = L(B(A)) \text{ and } \Theta(L(B(A))) = L_{TDS}(A).$$

Proof. We start by proving the leftmost equality. Let $r = r_0, r_1, \cdots$ be a stream of records in $L(B(A)) \subseteq Rec_{\mathcal{N}}(\mathcal{D})^{\omega}$. Because $B(A)$ is a Büchi automaton all whose states are final,

there is an infinite computation $\pi = q_0, r_0, q_1, r_1, \cdots$ in $B(A)$, starting from an initial state q_0 and where each tuple (q_i, r_i, q_{i+1}) is a transition in $B(A)$. By construction, for each transition (q_i, r_i, q_{i+1}) in the Büchi automaton $B(A)$, there is a transition (q_i, N_i, g_i, q_{i+1}) in the constraint automaton A, with a data assignment $\delta_i{:}N_i \rightarrow \mathcal{D}$ such that $\delta \models g_i$ and $\forall n \in N_i, r.n = \delta(n)$. This implies that the stream $\pi' = q_0, (N_0, g_0), q_1, (N_1, g_1), \cdots$ is an infinite computation in the constraint automaton A and that for all TDS-tuples $\theta \in TDS^{\mathcal{N}}$ with $r = \Upsilon(\theta)$ it holds that $\theta.N(i) = N_i$ and $\theta.\delta(i) \models g_i$, for all $i \geq 0$. Thus, $r \in \Upsilon(L_{TDS}(A))$ and $L(B(A)) \subseteq \Upsilon(L_{TDS}(A))$.

Conversely, let $r = r_0, r_1, \cdots$ be a stream of records in $\Upsilon(L_{TDS}(A))$. Then there is a TDS-tuple $\theta \in L_{TDS}(A)$ such that $r = \Upsilon(\theta)$ and for each $n \in \theta.N(k)$ and $k \in \mathbb{N}$, $r(k).n = \theta.\delta(k)_n$. Because $\theta \in L_{TDS}(A)$, there is a computation $\pi = q_0, (N_0, g_0), q_1, (N_1, g_1), \cdots$ in the constraint automaton A, starting from an initial state q_0 where $\theta.N(i) = N(i)$ and $\theta.\delta(i) \models g_i$, for all $i \geq 0$. By construction, there is a computation $\pi = q_0, r_0, q_1, r_1, \cdots$ in $B(A)$ and data assignments $\delta_i{:}N \rightarrow \mathcal{D}$ such that, for all $i \geq 0$, $\delta_i \models g_i$ and $r_i.n = \delta_i(n)$. Since in $B(A)$ all infinite runs starting from an initial state are accepting, $r \in L(B(A))$, and hence $\Upsilon(L_{TDS}(A)) \subseteq L(B(A))$.

Next we prove the rightmost equality. Let $\theta \in TDS^{\mathcal{N}}$ be a timed data stream accepted by the constraint automaton A, that is $\theta \in L_{TDS}(A)$. By definition of acceptance, there exists an infinite computation $\pi = q_0, (N_0, g_0), q_1, (N_1, g_1), \cdots$ in A such that, $q_0 \in Q_0$ and, for all $i \geq 0$, $(q_i, (N_i, g_i), q_{i+1})$ is a transition in A, $N_i = \theta.N(i)$, and $\theta.\delta(i) \models g_i$. But then, by construction, there is an infinite computation $\pi' = q_0, r_0, q_1, r_1, \cdots$ in the Büchi automaton $B(A)$ such that for all $i \geq 0$, there is a data assignment $\delta_i{:}N \rightarrow \mathcal{D}$ such that $\delta_i \models g$ and $\forall n \in N, r_i.n = \delta(n)$. Thus, $r = r_0, r_1, \cdots \in L(B(A))$ and $\theta = \Theta(r)$. Therefore, $L_{TDS}(A) \subseteq \Theta(L(B(A)))$.

Conversely, let $\theta \in TDS^{\mathcal{N}}$ be such that $\theta \in \Theta(L(B(A)))$. Then there is a stream of records $r = r_0 r_1 \cdots \in L(B(A))$, with $\theta = \Theta(r)$, that is, for all $k \geq 0$, $\theta.N(k) = dom(r_k)$ and $\forall n \in dom(r_k)$, $\theta.\delta(k)_n = r_k.n$. Because $r \in L(B(A))$, there is an infinite computation $\pi = q_0, r_0, q_1, r_1, \cdots$ in $B(A)$ with $q_0 \in Q_0$ and such that for all $i \geq 0$, the triple (q_i, r_i, q_{i+1}) is a transition in $B(A)$. By the construction of the Büchi automaton $B(A)$ from the constraint automaton A, there is an infinite computation $\pi' = q_0, (N_0, g_0), q_1, (N_1, g_1), \cdots$ in A such that for all $i \geq 0$, there is a data assignment $\delta_i{:}N \rightarrow \mathcal{D}$ which $\delta_i \models g_i$ and $\forall n \in N_i, r_i.n = \delta(n)$. Thus, $\theta = \Theta(r)$ and $\theta \in L_{TDS}(A)$. Therefore, $\Theta(L(B(A))) \subseteq L_{TDS}(A)$. □

It follows that Büchi automata of records are at least as expressive as constraint automata. They are actually more expressive, because Büchi automata of records are closed under (language) complement while constraint automata are not.

4.4 Modeling Fair Reo Connectors

As we mentioned in the introduction, for several connectors we can consider some fairness conditions. In this section, we present some useful *fair* connectors that can be modeled by

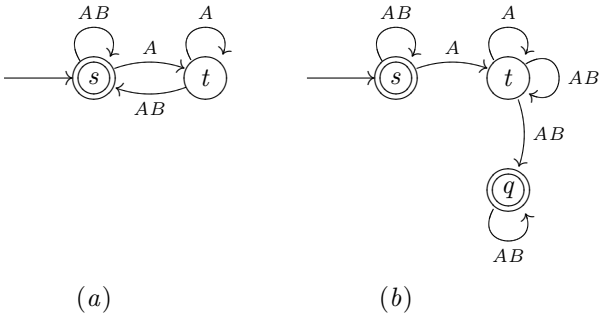


Figure 4.9: Models of a fair non-deterministic lossy synchronous channel with a) a weak fairness condition, b) a strong fairness condition.

Büchi automata of records.

Example 4.16 Consider the connector (over a singleton data domain) between two ports A and B with the behavior described by the Büchi automaton of records in Figure 4.9.a. It is a connector similar to the non-deterministic lossy synchronous channel depicted in Figure 4.8.b but with this extra property that not all data can get lost. Still infinitely many data can get lost, while the non-deterministic lossy synchronous channel modeled by Büchi automaton of records in Figure 4.9.b allows for loosing only *finitely many* data at the port A.

Because Büchi automata of records are Büchi automata, we can express *unconditional fairness conditions* [101]: in each infinite execution of the system, some actions should occur infinitely many times.

Example 4.17 Consider the *merger* connector with two source ports A and B and one sink port B (see Figure 4.6(a)). Intuitively, it transmits synchronously a data item from either A or B to the port C. If both the source ports A and B offer data at the same time then only one of them is chosen non-deterministically. The Büchi automaton of records over the data set $\mathcal{D} = \{d\}$ corresponding to the constraint automaton model of merger introduced in [30] is shown in Figure 4.10(a). Both models allow unfair executions where data from the same source is always preferred if both A and B always offer data simultaneously. Figure 4.10(b) shows a Büchi automaton that disallows those unfair executions. Because constraint automata do not distinguish between accepting and non-accepting states, they cannot express this kind of fairness conditions [30].

4.5 Composition of Büchi Automata of Records

Complex component connectors can be obtained by composing simpler ones, and by hiding some ports from the environment. Below we describe these operators on BARs. We will give few examples in the following section.

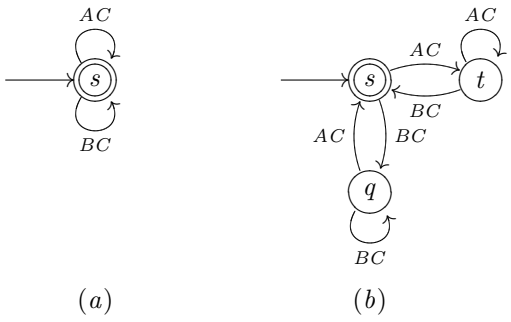


Figure 4.10: Models of a merger connector: (a) unfair version, (b) fair version

4.5.1 Product and Join

Since BARs are ordinary Büchi automata, we can compose them by means of the standard (synchronous) product for Büchi automata, provided they act on the same alphabet. The intuitive meaning of the product is the synchronization of the two component connectors they represent.

Recall the definition of the product of Büchi automata which, for simplicity, we give in terms of generalized Büchi automata as defined in Definition 4.6:

Definition 4.10 Let $B_1 = \langle Q_1, \Sigma, \longrightarrow_1, Q_{01}, F_1 \rangle$ and $B_2 = \langle Q_2, \Sigma, \longrightarrow_2, Q_{02}, F_2 \rangle$ be two Büchi automata on the same alphabet. The product of B_1 and B_2 is the *generalized* Büchi automaton:

$$B_1 \times B_2 = \langle Q_1 \times Q_2 . \Sigma, \longrightarrow, Q_{01} \times Q_{02}, \{F_1 \times Q_2, Q_1 \times F_2\} \rangle$$

where the transition relation $\longrightarrow$ is defined as:

$$\frac{q \xrightarrow{a}_1 q' \quad p \xrightarrow{a}_2 p'}{\langle q, p \rangle \xrightarrow{a} \langle q', p' \rangle}.$$

The language of the product of two Büchi automata is the intersection of their respective languages [138].

Note that the product of two such automata is a *generalized* Büchi automaton. To obtain an ordinary Büchi automaton for the product, one can use the fact that for each generalized Büchi automaton B there is an ordinary Büchi automaton B' such that $L(B) = L(B')$ (see Lemma 4.2) .

Join Using the richer structure of the alphabet of Büchi automata of records, we can give a more general definition of product that works even if the alphabets of the two automata are different.

Definition 4.11
Let $B_1 = \langle Q_1, Rec_{\mathcal{N}_1}(\mathcal{D}), \longrightarrow_1, Q_{01}, F_1 \rangle$ and $B_2 = \langle Q_2, Rec_{\mathcal{N}_2}(\mathcal{D}), \longrightarrow_2, Q_{02}, F_2 \rangle$ be two

BARs. We define the *join* of B_1 and B_2 as the *generalized* Büchi automaton $B_1 \bowtie B_2$ given by:

$$B_1 \bowtie B_2 = \langle Q_1 \times Q_2, Rec_{\mathcal{N}_1 \cup \mathcal{N}_2}(\mathcal{D}), \longrightarrow, Q_{01} \times Q_{02}, \{F_1 \times Q_2, Q_1 \times F_2\} \rangle$$

where the transition relation $\longrightarrow$ is defined by the following rules:
Rule 1:

$$\frac{q \xrightarrow{r_1}_1 q' \quad p \xrightarrow{r_2}_2 p' \quad comp(r_1, r_2)}{\langle q, p \rangle \xrightarrow{r_1 \cup r_2} \langle q', p' \rangle},$$

Rule 2:

$$\frac{q \xrightarrow{r_1}_1 q' \quad dom(r_1) \cap \mathcal{N}_2 = \emptyset}{\langle q, p \rangle \xrightarrow{r_1} \langle q', p \rangle},$$

and dually,

$$\frac{p \xrightarrow{r_2}_2 p' \quad dom(r_2) \cap \mathcal{N}_1 = \emptyset}{\langle q, p \rangle \xrightarrow{r_1} \langle q, p' \rangle}.$$

where by the proposition $comp(r_1, r_2)$ we mean that records r_1 and r_2 are compatible (see Definition 4.3).

Intuitively, in the join operation, two transitions are synchronized if they are labeled by compatible records (i.e. on the common ports they communicate the same data values), whereas they are interleaved if they are labeled with records not referring to ports of the other automaton.

Example 4.18 For example, consider Figure 4.11. Figure 4.11(a) shows the Büchi automaton of records modeling a FIFO1 channel between ports A and B (using as data set $\mathcal{D} = \{d\}$) and (b) a FIFO1 between ports B and C over the same data set. The join of these two automata is shown in Figure 4.11(c).

For Büchi automata without τ-transitions, the join operator coincides with the product in case both automata have the same alphabet. In this case, the language of the product is just the intersection of the languages of the two automata.

Lemma 4.5 Let B_1 and B_2 be two Büchi automata of records with the same alphabet $\Sigma = Rec_{\mathcal{N}}(\mathcal{D})$ (over the same data sets and the same port sets). Then,

$$L_{vis}(B_1 \bowtie B_2) = L_{vis}(B_1) \cap L_{vis}(B_2).$$

Proof. Let B_1' and B_2' be BARs without τ-transitions, respectively, the visibly equivalents of B_1 and B_2 after applying the τ-transitions elimination procedure that we introduced in the proof of Lemma 4.3. We know that for $i \in \{1, 2\}$, $L_{vis}(B_i) = L(B_i')$. Thus, it is enough to show that $L_{vis}(B_1 \bowtie B_2) = L(B_1' \times B_2')$.

Let $\omega \in L(B_1' \times B_2')$. Thus, ω has no τ symbol and $\omega \in L(B_1') \cap L(B_2')$. Because there is an accepting computation for ω in B_1', there is an infinite word $\rho_1 \in \Sigma^\omega$ such that $\rho_1 \in L(B_1)$ and $\omega = vis(\rho_1)$. Similarly, there is an infinite word $\rho_2 \in \Sigma^\omega$ such that $\rho_2 \in L(B_1)$ and $\omega = vis(\rho_2)$. Because ρ_1 and ρ_2 are visibly equivalent (namely, ignoring all τ symbols, both

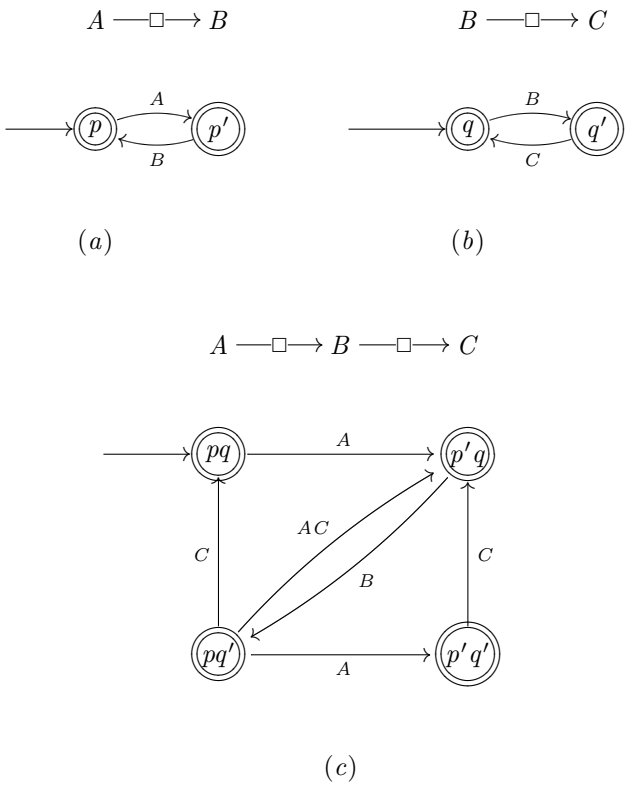


Figure 4.11: Composing two FIFO1 channels

become the same infinite word) and both are over the same alphabet, the first rule of the join operation (see Rule 1 in Definition 4.11) is applicable only on two τ-transitions or two transitions with the exact same labels (a τ-transition can not be synchronized with a transition with a label other than τ). Similarly, Rule 2 is applicable only on τ-transitions. Thus, there is an accepting infinite word $\rho_3 \in L(B_1 \bowtie B_2)$ such that $vis(\rho_3) = vis(\rho_2) = vis(\rho_1) = \omega$. Therefore, $\omega \in L_{vis}(B_1 \bowtie B_2)$.

Conversely, suppose that $\omega \in L_{vis}(B_1 \bowtie B_2)$. Thus, there is $\rho \in L(B_1 \bowtie B_2)$ such that $\omega = vis(\rho)$. Because B_1 and B_2 are over the same alphabets (same set of data and same set of names), again in the join operation, Rule 1 can be applied only on two τ-transitions or on two transitions with the exact same labels (a τ-transition cannot be synchronized with a transition with a label other than τ) and Rule 2 is applicable only on τ-transitions. Thus, there is an infinite word, say $\rho_1 \in L(B_1)$, that $vis(\rho_1) = vis(\rho)$. Similarly, there is an infinite word, say $\rho_2 \in L(B_2)$, that $vis(\rho_2) = vis(\rho)$. Thus, $\omega \in L(B_1') \cap L(B_2')$. Therefor, $\omega \in L(B_1' \times B_2')$. □

This implies that our definition of join is correct with respect to the product of ordinary Büchi automata (up to τ-transitions). On the other hand, our definition of join is correct (even

structurally, and not only language theoretically) also with respect to the join of constraint automata.

Theorem 4.6 Let A_1 and A_2 be two constraint automata. Then,

$$B(A_1) \bowtie B(A_2) = B(A_1 \bowtie_C A_2).$$

Proof. Let $A_1 = \langle Q_1, \mathcal{N}_1, T_1, Q_{01}\rangle$ and $A_2 = \langle Q_2, \mathcal{N}_2, T_2, Q_{02}\rangle$. Using Definition 3.8

$$A_1 \bowtie_C A_2 = \langle Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, T, Q_{01} \times Q_{02}\rangle,$$

where T is the set of all transitions obtained using rules presented in Definition 3.8. Using Definition 4.9, $B(A_1 \bowtie_C A_2)$ is

$$\langle Q_1 \times Q_2, Rec_{\mathcal{N}_1 \cup \mathcal{N}_2}(\mathcal{D}), \Delta_C, Q_{01} \times Q_{02}, Q_1 \times Q_2\rangle,$$

where Δ_C is the set of transitions $(\langle s, t\rangle, r, \langle s', t'\rangle)$ such that, there exists the transition $(\langle s, t\rangle, N, g, \langle s', t'\rangle) \in T$ and $\delta{:}N \to D$ such that $\delta \models g$ and for all n in N, $r.n = \delta(n)$.

Further, let

$$B(A_1) = \langle Q_1, Rec_{\mathcal{N}_1}(\mathcal{D}), \Delta_1, Q_{01}, Q_1\rangle \text{ and } B(A_2) = \langle Q_2, Rec_{\mathcal{N}_2}(\mathcal{D}), \Delta_2, Q_{02}, Q_2\rangle$$

with Δ_1 and Δ_2 obtained as described in Definition 4.9. Using Definition 4.11, $B(A_1) \bowtie B(A_2)$ is the automaton

$$\langle Q_1 \times Q_2, Rec_{\mathcal{N}_1 \cup \mathcal{N}_2}(\mathcal{D}), \Delta_B, Q_{01} \times Q_{02}, Q_1 \times Q_2\rangle$$

with Δ_B the set of all transitions obtained using the rules in Definition 4.11. We need to prove that $\Delta_C = \Delta_B$

First, we prove $\Delta_C \subseteq \Delta_B$. Let $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_C$. There is $(\langle s, t\rangle, N, g, \langle s', t'\rangle)$ in T and data assignment $\delta{:}N \to \mathcal{D}$, such that $\delta \models g$ and $\forall n \in N, r.n = \delta(n)$. We have three cases:

1) If $(\langle s, t\rangle, N, g, \langle s', t'\rangle) \in T$ is obtained using the first rule in Definition 3.8, then, there are $(s, N_1, g_1, s') \in T_1$ and $(t, N_2, g_2, t') \in T_2$ such that, $N = N_1 \cup N_2$, $N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1$, $\emptyset \neq N_1 \subseteq \mathcal{N}_1$ and $\emptyset \neq N_2 \subseteq \mathcal{N}_2$. Let $\delta|_{N_1}$ and $r|_{N_1}$ be respectively the restricted versions of δ and r for the domain $N_1 \subseteq N$. Obviously, $\delta|_{N_1} \models g_1$ and $\forall n \in N_1, r|_{N_1}.n = \delta|_{N_1}(n)$. Similarly, $\delta|_{N_2} \models g_2$ and $\forall n \in N_2, r|_{N_2}.n = \delta|_{N_2}(n)$. Thus, based on the definitions of Δ_1 and Δ_2, we conclude that $(s, r|_{N_1}, s') \in \Delta_1$ and $(t, r|_{N_2}, t') \in \Delta_2$. Because $r|_{N_1}$ and $r|_{N_2}$ both are restricted versions of r, $r|_{N_1}$ and $r|_{N_2}$ are compatible and $r|_{N_1} \cup r|_{N_2} = r$. Thus, using the first rule in Definition 4.11, $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_B$.

2) If $(\langle s, t\rangle, N, g, \langle s', t'\rangle) \in T$ is obtained using the second rule in Definition 3.8, then, there is $(s, N, g, s') \in T_1$ such that, $t = t'$ and $N \cap \mathcal{N}_2 = \emptyset$. Thus, based on the definition of Δ_1, we have $(s, r, s') \in \Delta_1$. Because $dom(r) \subseteq N$, therefore, $dom(r) \cap \mathcal{N}_2 = \emptyset$. Using Rule 2 in Definition 4.11, we conclude that $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_B$.

3) If $(\langle s, t\rangle, N, g, \langle s', t'\rangle) \in T$ is obtained using the third rule in Definition 3.8. The proof is similar to the previous case because the third rule is the dual of the second one.

It remains to prove $\Delta_B \subseteq \Delta_C$. Let $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_B$. We have three cases:

1) If $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_B$ is obtained using the first rule in Definition 4.11, then, there are

$(s, r_1, s') \in \Delta_1$ and $(t, r_2, t') \in \Delta_2$ such that, $r = r_1 \cup r_2$, records r_1 and r_2 are compatible, $dom(r_1) \cap \mathcal{N}_2 = dom(r_2) \cap \mathcal{N}_1$, $r_1 \neq \tau$ and $r_2 \neq \tau$. Thus, based on the definitions of Δ_1 and Δ_2, we conclude that there are $(s, N_1, g_1, s') \in T_1$ and $(t, N_2, g_2, t') \in T_2$ and data assignments $\delta_1{:}N_1 \rightarrow \mathcal{D}$ and $\delta_2{:}N_2 \rightarrow \mathcal{D}$ such that $\delta_1 \models g_1$, $\delta_2 \models g_2$, $\forall n \in N_1, r.n = \delta_1(n)$ and $\forall n \in N_2, r.n = \delta_2(n)$. Let $N = N_1 \cup N_2$, $g = g_1 \wedge g_2$ and $\delta = \delta_1 \cup \delta_2$. Because $dom(r_1) \cap \mathcal{N}_2 = dom(r_2) \cap \mathcal{N}_1$, therefore, $N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1$ and using the first rule in Definition 3.8, we have $(\langle s, t\rangle, N, g, \langle s', t'\rangle) \in \Delta_B$. Obviously, $\delta \models g$ and $\forall n \in N, r.n = \delta(n)$. Thus, by construction $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_C$.
2) If $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_B$ is obtained using the second rule in Definition 4.11, then, there is a $(s, r, s') \in \Delta_1$ such that $t = t'$ and $dom(r) \cap \mathcal{N}_2 = \emptyset$. Based on the definition of Δ_1, there is a $(s, N, g, s') \in T_1$ and there are data assignments $\delta{:}N \rightarrow \mathcal{D}$ such that $\delta \models g$ and $\forall n \in N, r.n = \delta(n)$. Because $dom(r) \cap \mathcal{N}_2 = \emptyset$, $N \cap \mathcal{N}_2 = \emptyset$ and using the second rule in Definition 3.8, we have $(\langle s, t\rangle, N, g, \langle s', t'\rangle) \in \Delta_B$. Thus,$(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_C$.
3) Again, the remaining case can be treated similarly. □

4.5.2 Splitting the Join

Next, we give an alternative way to calculate the join of two Büchi automata of records. The idea is to use the standard product after we have extended the alphabets of the two automata to a minimal common alphabet. First of all we concentrate on how to extend a Büchi automaton of records B with an extra port name, not necessarily present in the alphabet of B. If the port is new, the resulting automaton will have to guess the right behavior non-deterministically, by allowing or not the simultaneous exchange of data with the other ports known to the automaton.

Definition 4.12 Let $B = \langle Q, Rec_{\mathcal{N}}(\mathcal{D}), \Delta, Q_0, F\rangle$ be a Büchi automaton of records and n be a (port) name. We define the extension of B with respect to n as the following Büchi automaton of records:

$$B{\uparrow}n = \langle Q, Rec_{\mathcal{N}\cup\{n\}}(\mathcal{D}), \widehat{\Delta}, Q_0, F\rangle$$

where $\widehat{\Delta} = \Delta$ if $n \in \mathcal{N}$ and otherwise

$$\widehat{\Delta} = \Delta \cup \{(q, [n = d], q) | q \in Q, d \in \mathcal{D}\} \cup \{(q, r[n{:}= d], q') | (q, r, q') \in \Delta, d \in \mathcal{D}\}.$$

Note that in forthcoming discussions and proofs sometimes we refer to the second and third component-sets of $\widehat{\Delta}$ by Δ' and Δ''. Namely, we define $\widehat{\Delta} = \Delta \cup \Delta' \cup \Delta''$ where,

$$\Delta' = \{(q, [n = d], q) | q \in Q, d \in \mathcal{D}\}$$

and

$$\Delta'' = \{(q, r[n{:}= d], q') | (q, r, q') \in \Delta, d \in \mathcal{D}\}.$$

Intuitively, to extend Büchi automaton of records B with one extra port name n, we use the same structure of B and add only some new transitions to it representing the guesses of the new behavior of the automaton with respect to the new port n. There are three kinds of guess: the environment does not use the name n in a communication (explaining why $\Delta \subseteq \widehat{\Delta}$); or the environment uses the name n for a communication but no other port of B is

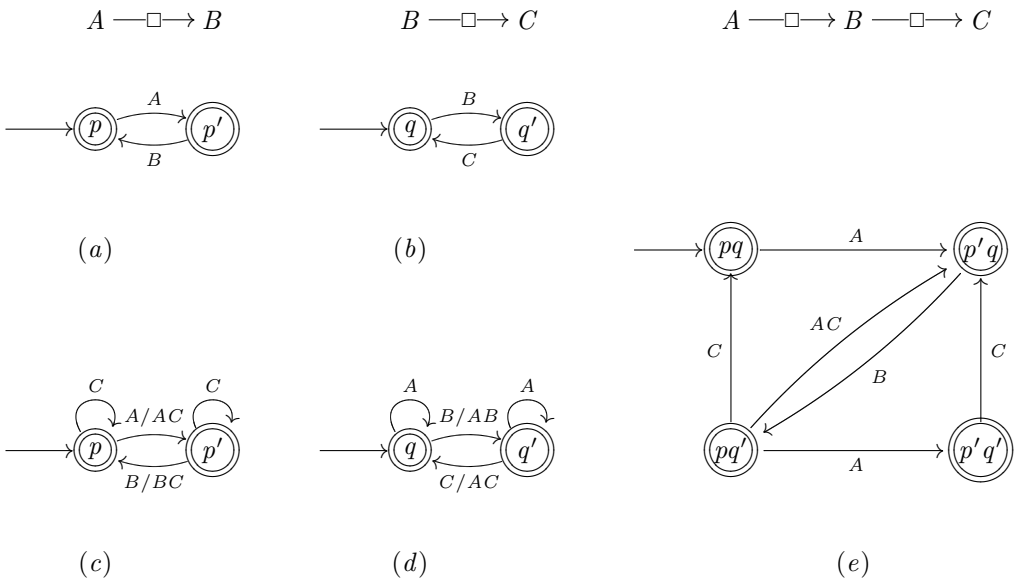


Figure 4.12: Direct and indirect joining of two FIFO1 buffers

used (explaining the addition of a new loop transition on each state labeled by a record with n as its only name in the domain); or the environment uses the name n in combination with the name constrained by B (corresponding to the new transitions of the form (q,r[n:=d],q') in Δ'. Recall here that $r[n{:=}\,d]$ is the extension of record r by adding the new field $n = d$ to it).

Example 4.19 For example, in Figure 4.12(c) we show the extension of the automaton has been shown in Figure 4.12(a) with respect to the new port name C. In this figure, $p \xrightarrow{A/AC} p'$ means that there are two transitions $p \xrightarrow{A} p'$ and $p \xrightarrow{AC} p'$. Also, Figure 4.12(d) is the extension of Figure 4.12(b) with A.

The operation of name extension is not sensitive to the order of different applications, in the sense that $(B{\uparrow}n){\uparrow}m = (B{\uparrow}m){\uparrow}n$, for two names n and m. Therefore, we can define the extension of a Büchi automaton with respect to a finite set of names N, denoted by $B{\uparrow}N$ by inductively extending the automaton B by one name in N at a time.

Given two Büchi automata of records B_1 and B_2 we can extend each of them with respect to the port names of the other, so that they become two Büchi automata over the same alphabet. We can thus take their ordinary product, obtaining as the result of the join of the two Büchi automata B_1 and B_2.

Theorem 4.7 Let B_1 and B_2 be two Büchi automata of records over alphabet sets $Rec_{\mathcal{N}_1}(\mathcal{D})$ and $Rec_{\mathcal{N}_2}(\mathcal{D})$, respectively. Then,

$$B_1{\uparrow}\mathcal{N}_2 \times_B B_2{\uparrow}\mathcal{N}_1 = B_1 \bowtie_B B_2.$$

Proof.
Let $B_1 = \langle Q_1, Rec_{\mathcal{N}_1}(\mathcal{D}), \Delta_1, Q_{01}, F_1\rangle$ and $B_2 = \langle Q_2, Rec_{\mathcal{N}_2}(\mathcal{D}), \Delta_2, Q_{02}, F_2\rangle$. Using Definition 4.11, $B_1 \bowtie B_2$ is

$$\langle Q_1 \times Q_2, Rec_{\mathcal{N}_1 \cup \mathcal{N}_2}(\mathcal{D}), \Delta_{\bowtie}, Q_{01} \times Q_{02}, F\rangle ,$$

where $F = \{F_1 \times Q_2, Q_1 \times F_2\}$ and $\Delta_{\bowtie}$ is the transition relation. Based on Definition 4.12, we have

$$B_1{\uparrow}\mathcal{N}_2 = \langle Q_1, Rec_{\mathcal{N}_1 \cup \mathcal{N}_2}(\mathcal{D}), \widehat{\Delta_1}, Q_{01}, F_1\rangle$$

and

$$B_2{\uparrow}\mathcal{N}_1 = \langle Q_2, Rec_{\mathcal{N}_1 \cup \mathcal{N}_2}(\mathcal{D}), \widehat{\Delta_2}, Q_{02}, F_2\rangle$$

where $\widehat{\Delta_1}$ and $\widehat{\Delta_2}$ are the transition relations. Their product is the Büchi automaton $B_1{\uparrow}\mathcal{N}_2 \times B_2{\uparrow}\mathcal{N}_1$ given by

$$\langle Q_1 \times Q_2, Rec_{\mathcal{N}_1 \cup \mathcal{N}_2}(\mathcal{D}), \Delta_{\times}, Q_{01} \times Q_{02}, F\rangle$$

where $\Delta_{\times}$ is defined according to Definition 4.10. We need to prove $\Delta_{\times} = \Delta_{\bowtie}$. We start by showing that $\Delta_{\times} \subseteq \Delta_{\bowtie}$:
Let $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_{\times}$. Using Definitions 4.10 and 4.12 we have,

$$(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_{\times} \iff (s, r, s') \in \widehat{\Delta_1} \wedge (t, r, t') \in \widehat{\Delta_2}$$

$$\iff (s, r, s') \in \Delta_1 \cup \Delta_1' \cup \Delta_1'' \wedge (t, r, t') \in \Delta_2 \cup \Delta_2' \cup \Delta_2''$$

We need to consider nine different cases:
1) $(s, r, s') \in \Delta_1$ and $(t, r, t') \in \Delta_2$. Obviously, using the first rule in Definition 4.11, we have $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_{\bowtie}$.
2) $(s, r, s') \in \Delta_1$ and $(t, r, t') \in \Delta_2'$. By the definition of Δ_2', $t = t'$ and $r \in Rec_{\mathcal{N}_1 \setminus \mathcal{N}_2}(\mathcal{D})$. Thus, $dom(r) \cap \mathcal{N}_2 = \emptyset$. Therefore, using the second rule in Definition 4.11, $(\langle s, t\rangle, r, \langle s', t\rangle)$ is in $\Delta_{\bowtie}$.
3) $(s, r, s') \in \Delta_1$ and $(t, r, t') \in \Delta_2''$. According to the definition of Δ_2'', there is a $(t, r', t') \in \Delta_2$ such that $dom(r) = dom(r') \cup N'$ for some $N' \subseteq \mathcal{N}_1 \setminus \mathcal{N}_2$ and $\forall n \in dom(r'){:}r(n) = r'(n)$. Therefore, $dom(r) \cap \mathcal{N}_2 = dom(r') \cap \mathcal{N}_1 = dom(r')$, r and r' are compatible and $r \cup r' = r$. Thus, using Definition 4.11 Rule 1, $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_{\bowtie}$.
4) $(s, r, s') \in \Delta_1'$ and $(t, r, t') \in \Delta_2$. The proof of this case is symmetric to the proof of case 2.
5) $(s, r, s') \in \Delta_1'$ and $(t, r, t') \in \Delta_2'$. This case is impossible, because, by the definition of Δ_1', $dom(r) \subseteq \mathcal{N}_2 \setminus \mathcal{N}_1$ and by definition of Δ_2', $dom(r) \subseteq \mathcal{N}_1 \setminus \mathcal{N}_2$ and $dom(r) \neq \emptyset$. Obviously, these conditions are contradictory.
6) $(s, r, s') \in \Delta_1'$ and $(t, r, t') \in \Delta_2''$. This case is impossible. Its proof is similar to case 5.
7) $(s, r, s') \in \Delta_1''$ and $(t, r, t') \in \Delta_2$.The proof of this case is symmetric to the proof of case 3.
8) $(s, r, s') \in \Delta_1''$ and $(t, r, t') \in \Delta_2'$. This case is impossible. Its proof is similar to case 5.
9) $(s, r, s') \in \Delta_1''$ and $(t, r, t') \in \Delta_2''$. According to the definition of Δ'', there are records r' and r'' such that $dom(r) = dom(r') \cup N' = dom(r'') \cup N''$ for $N' \subseteq \mathcal{N}_2 \setminus \mathcal{N}_1$ and $N'' \subseteq \mathcal{N}_1 \setminus \mathcal{N}_2$. By a simple set theoretic justification, it can be shown that, $dom(r') \cap \mathcal{N}_2 = dom(r'') \cap \mathcal{N}_1$ and because $\forall n \in dom(r'){:}r(n) = r'(n)$ and $\forall n \in dom(r''){:}r(n) = r''(n)$,

we have $r = r' \cup r''$. Thus, using Definition 4.11, Rule 1, $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_{\bowtie}$.

Next we prove that $\Delta_{\bowtie} \subseteq \Delta_{\times}$. Let $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_{\bowtie}$. We have two cases:
1) If $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_{\bowtie}$ is obtained using the first rule of Definition 4.11, there are $(s, r_1, s') \in \Delta_1$ and $(t, r_2, t') \in \Delta_2$ such that r_1 and r_2 are compatible, $r = r_1 \cup r_2$ and $dom(r_1) \cap \mathcal{N}_2 = dom(r_2) \cap \mathcal{N}_1$. Obviously, $(s, r, s') \in \Delta_1''$ and $(t, r, s') \in \Delta_1''$. Thus, $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_{\times}$.
2) If $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_{\bowtie}$ is obtained using the second rule of Definition 4.11, there is a $(s, r, s') \in \Delta_1$ such that $dom(r) \cap \mathcal{N}_2 = \emptyset$ and $t = t'$. Because $r \in Rec_{\mathcal{N}_1}(\mathcal{D})$ and $dom(r) \cap \mathcal{N}_2 = \emptyset$, we have $r \in Rec_{\mathcal{N}_1 \setminus \mathcal{N}_2}(\mathcal{D})$. Based on the definition of Δ', $(t, r, t) \in \Delta_2'$. Thus $(s, r, s') \in \widehat{\Delta_1}$ and $(t, r, t') \in \widehat{\Delta_2}$. Therefore, using the definition of Büchi product, $(\langle s, t\rangle, r, \langle s', t'\rangle) \in \Delta_{\times}$. □

Therefore, to join two Büchi automata of records, one can first extend them to a common set of ports and then compose the resulting Büchi automaton using the standard Büchi product operation. Based on the previous theorem, the automata produced by both methods are structurally, and thus also language theoretically, the same.

Example 4.20 The join of the Büchi automata of records shown in Figures 4.12(a) and (b) is the automaton shown in 4.12(e). This automaton, in turn, is the *product* of the automata depicted in Figures 4.12(c) and 4.12(d). The resulting automaton models a two-cell queue. Note that one of the diagonal transitions corresponds to the move of data from one cell to the other, while the other diagonal models the simultaneous consumption of data from port C and the insertion of a new data item through the port A.

4.5.3 Hiding of Port Names

The effect of hiding a port of a component connector is that data flow through that node is no longer observable. In BARs, the hiding operator removes all information about the hidden port.

Definition 4.13 Let $B = \langle Q, Rec_{\mathcal{N}}(\mathcal{D}), \rightarrow, Q_0, F\rangle$ be a BAR or generalized BAR. The hiding of a port name $A \in \mathcal{N}$ from B is the following BAR or generalized BAR:

$$B{\downarrow_A} = \langle Q, Rec_{\mathcal{N}\setminus\{A\}}(\mathcal{D}), \longrightarrow', Q_0, F\rangle$$

where $q \overset{r\setminus A}{\longrightarrow}' p$ if and only if $q \overset{r}{\longrightarrow} p$.

Note that if the domain of a record labeling a transition contains only the name to be hidden, then the transition becomes an internal one. It is easy to verify that (visibly) language equivalence is a congruence with respect to join and hiding.

The hiding operation is interesting when it is used after joining the Büchi automata of records that model some Reo connectors. In such cases, we are generally interested to hide the common or intermediate port names. In other words, by joining of connectors, we normally construct more complicated connectors in which the common ports of the elementary connectors become internal nodes and the other ports become the interfaces of the new connector.

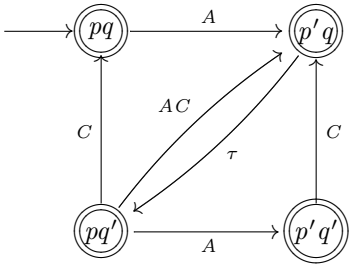


Figure 4.13: The resulting BAR after hiding B in Figure 4.12(e).

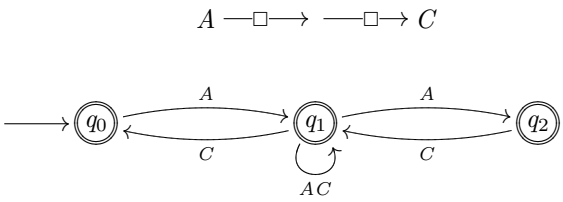


Figure 4.14: The resulting BAR after eliminating τ-transitions in Figure 4.13.

Thus, after joining of connectors, we can hide the effects of the common (intermediate) ports. Let us have an example:

Example 4.21 Figure 4.12(e) shows a BAR which is the resulting automaton after joining the two automata models of two FIFO1 channels (one from port A to port B and the other from port B to port C) that were illustrated in Figures 4.12(a) and (b). Now, B is an intermediate port. We hide port name B in the BAR illustrated in Figure 4.12(e). The resulting automaton is illustrated in Figure 4.13.

Now, we convert the generalized BAR model in Figure 4.13 to an equivalent BAR without τ-transition using the constructions that we introduced in the proofs of Lemma 4.2 (to convert generalized BAR to ordinary BAR) and Lemma 4.3 (to eliminate τ-transitions). The reachable part of the resulted automaton after conversion is illustrated in Figure 4.14.

As we expect, Figure 4.14 is a model of the visible behavior of a FIFO channel whose buffer capacity is two (say, FIFO2). States q_0, q_1 and q_2, respectively, represent the configurations of the connector where the buffer is empty, the buffer contains one data item, and the buffer is full.

4.6 Fair Constraint Automata

As we explained earlier, the timed data streams based semantics of constraint automata is more concrete than necessary. On the other hand, languages of streams of records that we have used as the semantics of BARs are more understandable and a more suitable semantics for connectors. In this section, we introduce a version of constraint automaton, called *fair*

constraint automaton, whose syntax is the same as constraint automaton except that now it has final (accepting) states, but its semantics is based on the languages of streams of records. As for the case of Büch automata, by adding sets of final states to constraint automata, *unconditional* fairness constraints over sets of states/transitions become expressible.

Definition 4.14 Let $\mathcal{D}$ be a fixed finite set of data. A *fair constraint automaton* (abbreviated as FCA) over a data set $\mathcal{D}$ is of the form $C = \langle Q, \mathcal{N}, \longrightarrow, Q_0, F \rangle$ where:

- Q is a finite set of states,
- $\mathcal{N}$ is a finite set of names,
- $\longrightarrow \subseteq\ Q \times 2^{\mathcal{N}} \times DC \times Q$ is a set of transitions, where, DC is the set of all data constraints over names set $\mathcal{N}$ and data set $\mathcal{D}$ as defined in Definition 3.5,
- $Q_0 \subseteq Q$ is the set of initial states,
- $F \subseteq Q$ is the set of accepting (final) states.

We write $p \xrightarrow{N,g} q$ instead of $\langle p, N, g, q \rangle \in \longrightarrow$ and call N the name set and g the guard of the transition.

A fair constraint automaton $C = \langle Q, \mathcal{N}, \longrightarrow, Q_0, F \rangle$ is *deterministic* if $|Q_0| \leq 1$ and for every state q, set of port names N, and data assignment $\delta{:}N \rightarrow \mathcal{D}$, there is at most one transition $q \xrightarrow{N,g} q'$ with $\delta \models g$.

Now, we define the semantics of FCAs using the languages of streams of records.

Definition 4.15 Let $C = \langle Q, \mathcal{N}, \longrightarrow, Q_0, F \rangle$ be an FCA over a data set $\mathcal{D}$. Also, suppose that $\Sigma = Rec_{\mathcal{N}}(\mathcal{D})$ and $\omega \in \Sigma^{\omega}$.
1- An *infinite computation* in C is an infinite sequence of alternating states and data constraints of the form $\pi = q_0, (N_0, g_0), q_1, (N_1, g_1), \ldots$ where $q_0 \in Q_0$ and for all subsequence $q_i, (N_i, g_i), q_{i+1}$ in π, $\langle q_i, N_i, g_i, q_{i+1} \rangle \in \longrightarrow$.
2- An infinite computation $\pi = q_0, (N_0, g_0), q_1, (N_1, g_1), \ldots$ is a *computation for the stream* $\omega = r_0 r_1 \ldots \in \Sigma^{\omega}$ if and only if for all $i \geq 0$, there is a data assignment $\delta{:}N_i \rightarrow \mathcal{D}$ such that $\delta \models g_i$, $dom(r_i) = N_i$ and $\forall n \in N_i{:}\ r.n = \delta(n)$.
3- Let $\pi = q_0, (N_0, g_0), q_1, (N_1, g_1), \ldots$ be a computation for the stream $\omega \in \Sigma^{\omega}$. We say that π is an *accepting computation* for ω if at least one of the accepting states, say $q_F \in F$, occurs infinitely many times in π.
4- We define the language of FCA C as follows:

$$L(C) = \{\omega \in Rec_{\mathcal{N}}(\mathcal{D})\omega \mid \text{there is an accepting computation } \pi \text{ for } \omega \text{ in } C.\}$$

5- Two FCAs C_1 and C_2 are equivalent if $L(C_1) = L(C_2)$.
6- The notions of *generalized fair constraint automata* (GFCA) and their accepted languages are defined similarly such as generalized Büchi automata and their accepted languages.

Obviously, because the semantics of both Büchi automaton of records and unconditional fair constraint automaton are based on languages of streams of records, each BAR B over

a name set $\mathcal{N}$ and a data set $\mathcal{D}$ can be considered as an FCA if we replace every transition label $r \in Rec_{\mathcal{N}}(\mathcal{D})$ with (N, g) where, $N = dom(r)$ and g is the data constraint $\bigwedge_{n \in dom(n)}(d_n = r.n)$. Using this simple conversion of BAR into FCA, we can show that all BARs that we introduced as models of Reo connectors can be considered as FCA models for them.

Conversely, if C is an FCA over a finite name set $\mathcal{N}$ and a finite data set $\mathcal{D}$ then C is equivalent with a BAR B all whose components are the same as C except that each transition $q \xrightarrow{N,g} q'$ of C is replaced with a set of transitions of the form $q \xrightarrow{r} q'$ where r satisfies the following conditions: $dom(r) = N$ and there exists a data assignment $\delta{:}N \to \mathcal{D}$ such that $\delta \models g$ and $\forall n \in N,\ \delta(n) = r.n$.

Every constraint automaton A can be converted to an FCA $C(A)$ all of whose states are accepting and with the same components as A. By this conversion, we have the following direct consequence:

Theorem 4.8 Let $A = \langle Q, \mathcal{N}, \longrightarrow, Q_0 \rangle$ be a finite constraint automaton over a data set $\mathcal{D}$. For its corresponding FCA $C(A) = \langle Q, \mathcal{N}, \longrightarrow, Q_0, F \rangle$ over the same data set $\mathcal{D}$ in which all shared components are the same and $F = Q$, we have:

$$\Upsilon(L_{TDS}(A)) = L(C(A)) \text{ and } \Theta(L(C(A))) = L_{TDS}(A).$$

Proof. Let B be the BAR all whose components are the same as $C(A)$ except that each transition $q \xrightarrow{N,g} q'$ of $C(A)$ is replaced with a set of transitions of the form $q \xrightarrow{r} q'$ where r satisfies the following conditions: $dom(r) = N$ and there exists a data assignment $\delta{:}N \to \mathcal{D}$ such that $\delta \models g$ and $\forall n \in N,\ \delta(n) = r.n$. Obviously, we have $L(B) = L(C(A))$, and using Definition 4.9, B is the corresponding BAR for A. Based on Theorem 4.4, $\Upsilon(L_{TDS}(A)) = L(B)$ and $\Theta(L(B)) = L_{TDS}(A)$. Thus, the result holds. □

Same as the case of constraint automata, fair constraint automata can be composed by a join operator:

Definition 4.16 Let $C_1 = \langle Q_1, \mathcal{N}_1, \longrightarrow_1, Q_{01}, F_1 \rangle$ and $C_2 = \langle Q_2, \mathcal{N}_2, \longrightarrow_2, Q_{02}, F_2 \rangle$ be two FCAs both over the set of data $\mathcal{D}$. The *join* of C_1 and C_2 is the GFCA:

$$C_1 \bowtie C_2 = \langle Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, \longrightarrow, Q_{01} \times Q_{02}, \mathcal{F} \rangle$$

where, the transition relation $\longrightarrow$ is exactly as defined for the join of constraint automata (see Definition 3.8) and the set of sets of accepting states is

$$\mathcal{F} = \{Q_1 \times F_2, F_1 \times Q_2\}.$$

Based on the above definition, we have the following theorem:

Theorem 4.9 Let $C_1 = \langle Q_1, \mathcal{N}, \longrightarrow_1, Q_{01}, F_1 \rangle$ and $C_2 = \langle Q_2, \mathcal{N}, \longrightarrow_2, Q_{02}, F_2 \rangle$ be two FCAs both over the same set of names $\mathcal{N}$ and the same set of data $\mathcal{D}$. Then,

$$L_{vis}(C_1 \bowtie C_2) = L_{vis}(C_1) \cap L_{vis}(C_2).$$

Proof. The theorem is a direct consequence of Lemma 3.1(2), Lemma 4.5, and Theorem 4.8. □

Also, the hiding operator over FCAs is the same as the hiding operator over constraint automata (see Definition 3.10).

5 Context Dependent Connectors

In the previous chapter we addressed one specific shortcoming of the constraint automata as a model of Reo networks, namely the impossibility to model desirable fairness constraints. In this chapter we address another deficiency of constraint automata, that is, their inability to model behavior that depends on pending I/O operations on the ports of a connector. This latter property is called *context dependency*, which manifests itself when the behavior of a connector can change depending upon not only the presence of requests on a connector boundary, but also on their absence.

5.1 Introduction

The prototypical Reo connector featuring a context depended behavior is the *context dependent* lossy synchronous channel (not to be confused with the previous non-deterministic and fair lossy synchronous channels): if the port connected at the source is ready to send data but the port at the sink is not ready to receive, then the data at the source is lost. Until now, we have ignored such requirements and lossy synchronous channels have been modeled by constraint automata or BARs using a (fair) non-deterministic choice. While this is sufficient for modeling Reo networks like the exclusive router presented in Figure 3.2, in general, the presence of context dependent lossy synchronous channels increase the expressiveness of Reo models [13].

First, we describe precisely our definition of the notion of context dependency (or context sensitivity). Context dependency means that the choice of the transition/behavior of a channel/system depends on the (un)availability of I/O requests on its ports. In a trivial sense, all automata/systems can be considered context dependent, because their choice of a transition, of course, depends on the availability of I/O requests on their ports. But, this overly general sense of context dependency is useless. So, we restrict the term *context dependency* to refer to only those cases where the behavior of a channel/system depends on the unavailability of I/O requests over its ports.

In order to address context dependent behavior, we extend the BAR models with the possibility of testing if some ports of the environment are ready to communicate or not. That is, we consider a Büchi variant of Kozen's finite automata on guarded strings [100]. In our case, an infinite guarded string is an alternating sequence of sets of *ready* ports and records of *fired* ports (together with their respective data-flow). The difficulty in correctly addressing a context dependent behavior is not in its modeling per se, but in its effect when composing different connectors. In fact, as for the combination of synchronous and mutual exclusion constraints, also context dependencies should propagate across a connector. This means that the models of two connectors when composed should agree on both the synchronized and mutually excluded common ports, as well as on the tests of the common ports. With this aim, we present a novel definition of a composition operator that generalizes the automata product construction by allowing the alphabets of the two automata to be different.

Our model, called *augmented Büchi automaton of records* (ABAR), has the advantage over previous models for Reo in that it covers the basic concepts of Reo as well as the context sensitive behavior within a standard automata theoretical framework. In fact, we show that

not only every BAR model of a connector can be transformed into an ABAR model but also the context dependent requirements can be modeled by ABARs. The benefits are a clear and easy notation for the representation of component connectors, as well as the efficient existing tool support for automatic analysis.

5.2 Guarded Languages and Augmented Büchi Automata

In this section we augment our model for component connectors so to take into account context dependencies like the ones of the *lossy synchronous channel*: if the port connected at the source is ready for accepting data but the port at the sink it is not ready for receiving it, then the data at the source is lost. In the previous chapter, we have ignored such a requirement and modeled the loss of data by means of a (fair) non-deterministic choice with a BAR. In this section, we extend Büchi automata of records with the capability of modeling coordination strategies based on pending and ignored ports. The idea is to enrich the states of a BAR automaton with expressions for testing if the ports shared with the environment are ready to communicate or not. Intuitively, a transition

$$q \xrightarrow{r} p$$

can be taken only if the ports of the system successfully pass the test associated with a state q. This implies that we must be able to safely eliminate states associated with tests that always fail, and that passing a test has to guarantee that at least as many ports are ready to communicate as needed by every outgoing transitions.

More formally, we consider the set $\mathcal{N}$ of port names as our *primitive test* symbols. Next, we define the set $Exp_{\mathcal{N}}$ of expression for Boolean tests for $\mathcal{N}$ as follows:

Definition 5.1 Let $\mathcal{N}$ be a set of port names. The set $Exp_{\mathcal{N}}$ of expression for Boolean tests for $\mathcal{N}$ is defined by the grammar

$$e ::= 1 \mid 0 \mid A \mid \bar{A} \mid e \cdot e$$

where $A \in \mathcal{N}$.

Each test expression $e \in Exp_{\mathcal{N}}$ is evaluated over a set $N \subseteq \mathcal{N}$ of ports (ready to communicate):

Definition 5.2 Given a set $N \subseteq \mathcal{N}$ of ports, we define when N passes the test expression e, denoted by $N \models e$, as follows:

$$\begin{array}{lll} N \models 1 & & \\ N \not\models 0 & & \\ N \models A & \text{iff} & A \in N \\ N \models \bar{A} & \text{iff} & A \notin N \\ N \models e_1 \cdot e_2 & \text{iff} & N \models e_1 \text{ and } N \models e_2 \end{array}$$

Informally, every collection of ports ready to communicate passes the test expression 1 while every collection of ports ready to communicate containing A passes the primitive test A. The conjunction of two tests e_1 and e_2 is the test $e_1.e_2$, while the negation of a primitive test A is denoted by $\bar{A}$. Note that while we use all test expressions in positive normal form, in general the negation can be used over every test expression, say e, using $\bar{e}$. Then the positive form can be obtained. In this case, the other boolean connectives can be defined as derived operators, for instance we define the disjunction of two expressions e_1 and e_2 to be given by $\overline{\bar{e_1}.\bar{e_2}}$. We use $\equiv$ to denotes the propositional logic equivalence on $Exp_{\mathcal{N}}$.

Now, we define when a record can be executed:

Definition 5.3 Given a record $r \in Rec_{\mathcal{N}}(\mathcal{D})$, let $wp(r)$ be the *weakest precondition* for r to be executed. It is defined inductively on the size of $dom(r)$ as the following expression (up to $\equiv$):

$$wp(\tau) = 1$$
$$wp(r) = A \cdot wp(r \setminus A) \quad \text{if } A \in dom(r)$$

Intuitively, the expression $wp(r)$ is a test checking if all the ports synchronized by r are ready to communicate. Thus, in this case, a transition labeled by r can be fired.

We are now ready to introduce our extension of BARs for modeling both synchronization and context dependencies.

Definition 5.4 An *augmented Büchi automaton of records* (abbreviated by ABAR) is a pair $\langle B, l\rangle$ consisting of a BAR $B = \langle Q, Rec_{\mathcal{N}}(\mathcal{D}), \rightarrow, Q_0, F\rangle$ and labeling function $l{:}Q \rightarrow Exp_{\mathcal{N}}$ such that for all $q \in Q$, if $q \xrightarrow{r} p$ then $l(q)$ implies $wp(r)$.

As a consequence of the above definition, if $l(q) = \bar{A}$, then all transitions outgoing from q must be internal, i.e., they must be labeled by τ. Similarly, all transitions outgoing from a state labeled by 1 must be internal.

We will define ABARs as acceptors of infinite guarded strings [85]. We define our notion of infinite guarded strings:

Definition 5.5 An infinite *guarded string* over the alphabet $Rec_{\mathcal{N}}(\mathcal{D})$ is an alternating infinite sequence $N_0 r_0 N_1 r_1 \cdots$ where $r_i \in Rec_{\mathcal{N}}(\mathcal{D})$ and each N_i is a subset of ports in $\mathcal{N}$. We define a guarded language over the alphabet $Rec_{\mathcal{N}}(\mathcal{D})$ as a set of infinite guarded strings over the same alphabet.

Intuitively, a guarded string represents an execution of the system, where for each step it records the ports ready for communication and the actual data flow among a subset of them. More formally,

Definition 5.6
(1) Let $\gamma = N_0 r_0 N_1 r_1 \cdots$ be an infinite guarded string over alphabet $Rec_{\mathcal{N}}(\mathcal{D})$. We define an *infinite computation* for γ in an ABAR $\langle B, l\rangle$ (over the same alphabet) to be an infinite sequence $\pi = q_0, r_0, q_1, r_1, ...$, of alternating states and records in which $q_0 \in Q_0$, $N_i \models l(q_i)$ and $q_i \xrightarrow{r_i} q_{i+1}$ for all $i \in \mathbb{N}$.
(2) An infinite guarded strings γ is *accepted* by ABAR $\langle B, l\rangle$ if there is an infinite computation for γ in $\langle B, l\rangle$ with at least one of the final states occurring infinitely often.
(3) The *guarded language* of an ABAR $\langle B, l\rangle$, denoted by $GL(B)$, is the set of all infinite guarded strings accepted by it.

Note that the condition of an ABAR $\langle B, l\rangle$ that for every state q, if $q \xrightarrow{r} p$ then $l(q)$ implies $wp(r)$ means that for every guarded string $N_0 r_0 N_1 r_1 \cdots$ accepted, $dom(r_i) \subseteq N_i$ for all $i \geq 0$.

Definition 5.7
(1) We say that two ABARs B_1 and B_2 are *guarded-language equivalent* if $GL(B_1) = GL(B_2)$.
(2) We say that ABAR B and BAR B' are *(language) equivalent* if after ignoring the labeling function of B and considering its language of infinite strings of record we have $L(B) = L(B')$.

Given an ABAR $\langle B, l\rangle$ we can construct a guarded-language equivalent ABAR $\langle B', l'\rangle$ such that $l'(q) \not\equiv 0$ for all states q of B'. In fact, we can safely delete these inconsistent states from the set of states of B and their incoming and outgoing transitions because no set of names N will ever pass the test 0 (not even the empty set of names).

An augmented Büchi automaton of records can be considered as a Büchi automaton of records, if we ignore the labeling function.

Conversely, every Büchi automaton of records B can be transformed into a canonical ABAR $\langle B, l\rangle$ by assigning to each state q of B the conjunction of all $wp(r)$ for each record r labeling outgoing transitions from q. Namely:

Definition 5.8 Let $B = \langle Q, Rec_{\mathcal{N}}(\mathcal{D}), Q_0, \rightarrow, F\rangle$ be a BAR. The *canonical ABAR* for B is the ABAR $\langle B, l\rangle$ where the labeling function is define as follows:

$$\forall q \in Q, l(q) = \bigwedge_{r \in W} wp(r)$$

in which r's are the members of the following set of records:

$$W = \{r \mid \exists q \xrightarrow{r} q' \in B\}.$$

If B be a BAR and B' be its canonical ABAR then considering their languages of streams of records they are equivalent. Let us to have an example:

Example 5.1 Consider the BAR model illustrated in Figure 5.1(a). In fact it is a model of a FIFO2 channel from port A to C obtained after joining two FIFO1 channels and hiding the intermediate port (see Example 4.21). The canonical ABAR for it is illustrated in Figure 5.1(b).

Transforming a BAR into its canonical ABAR and back produces the same BAR, while the converse holds only for an ABAR without states with negative tests.

Although ABARs are as expressive as BARs, in terms of the languages of records that they recognize, they are more concrete. We will use this extra information when composing them. For the moment, we observe that for an ABAR $\langle B, l\rangle$ we can give a formal definition of its pending and ignored ports. Given a set N of ports, we say that $A \in N$ is *ignored* by a transition $q \xrightarrow{r} p$ if $N \models l(q)$ but $A \notin dom(r)$, that is, the port A may be ready to communicate but it is excluded by r. Similarly, we say that a port A is *pending* in a state q if

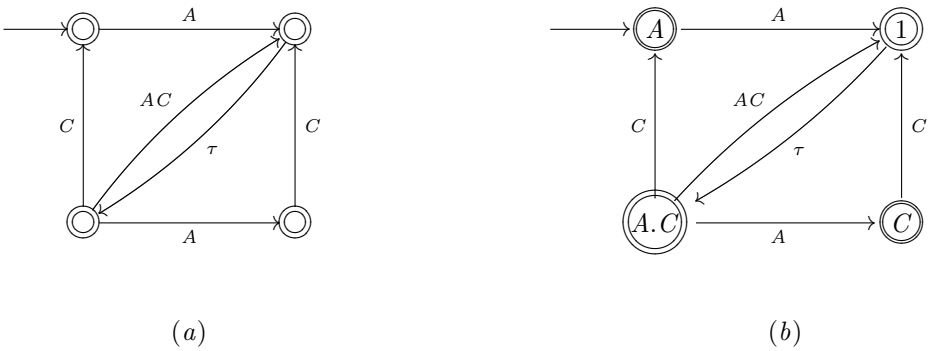
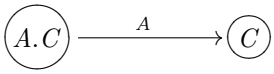


Figure 5.1: A BAR model of a FIFO2 channel and its canonical ABAR.

it is ignored by all transitions outgoing from q. For example, consider the ABAR illustrated in Figure 5.1(b). In the following transition, the port $C \in \{A, C\}$ has been ignored:



Also, suppose that B is an ABAR all whose components are the same as the ABAR illustrated in Figure 5.1(b) except that its initial state is $A \cdot C$. In this case, the port C is suspended in the initial state.

Definition 5.9 We say that two ABARs $\langle B_1, l_1 \rangle$ and $\langle B_2, l_2 \rangle$ are *visibly equivalent* if they have no state labeled by an expression logically equivalent with 0 and $L_{vis}(B_1) = L_{vis}(B_2)$.

Remark 5.1 Sometimes, it is more readable in the definition of an ABAR to assign a set of sets of port names as the label of a state, instead of using a boolean test expression as its label. In other words, based on Definition 5.2 each boolean test expression e can be interpreted as the set of all subsets of the set of port names that satisfy e. Thus, these sets can be directly assigned to the states as their labels. More formally, let $\mathcal{N}$ be the set of port names, $B = \langle Q, Rec_{\mathcal{N}}(\mathcal{D}), \rightarrow, F \rangle$ be a BAR and $\langle B, l \rangle$ be an ABAR using BAR B with a proper labeling function $l{:}Q \rightarrow Exp_{\mathcal{N}}$. We define a labeling function $V{:}Q \rightarrow (2^{\mathcal{N}} \rightarrow \{true, false\})$ such that:

$$\forall N \subseteq \mathcal{N}{:}\ \ V(q)(N) = true \text{ if and only if } N \models l(q).$$

We can semantically consider $\langle B, V \rangle$ as equivalent with the ABAR $\langle B, l \rangle$. In the next chapter, we will translate temporal formulas of our proposed temporal logic, called ρLTL, into ABARs of the form $\langle B, V \rangle$.

5.3 Modeling Reo connectors by ABARs

Now we present the ABAR models of basic Reo connectors and some other useful examples.

Example 5.2 Figure 5.2 shows three visibly equivalent ABAR models of the context dependent lossy synchronous channel from source port A to sink port B over a singleton data domain.

The ABAR illustrated in Figure 5.2(a), which is the most compact one, expresses that if both sink and source ports are ready to communicate simultaneously they exchange data. But if the source is ready while the sink is not the data will be lost. The ABAR model in Figure 5.2(b) expresses the same while it also models the state in which no port is ready. Finally, the ABAR model in Figure 5.2(c) not only models the above mentioned properties but it also allows the sink port to be *suspended* while the source port has no data to deliver or is not ready to communicate. Note that the behavior of a context dependent lossy synchronous channel (as an open system) is deterministic, in the sense that, there is no scenario for the behavior of its environment that allows the channel to be able to make a choice between some transitions. Thus, all possible runs of the context dependent lossy synchronous channel are fair. Therefore, all states in Figures 5.2(a), (b) and (c) are accepting.

Now, to show that the ABAR model is able to express fairness conditions, we consider a closed system containing a context dependent lossy synchronous channel plus its environment, and require that in each trace of this system only finitely many times the input data into the channel can be lost. The enhanced ABAR model supporting this stronger fairness condition is shown in Figure 5.3. Similarly, the ABAR models of Figure 5.2(b) and (c) can be enhanced to support such fairness conditions.

Example 5.3 In Figure 5.4 we show the BAR and two ABAR models of a synchronous channel with source end B and sink C over a singleton data set. The model illustrated in Figure 5.4(b) is the canonical extension of the BAR model in Figure 5.4(a). Compare the expressiveness of the two ABAR models for synchronous channel. While the ABAR in Figure 5.4(b) accepts only the infinite guarded string

$$\{B, C\}[B = d, C = d]\{B, C\}[B = d, C = d]\cdots,$$

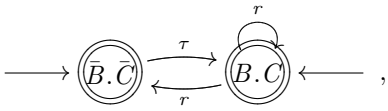
the automaton illustrated in Figure 5.4(c) accepts infinitely many strings, including

$$\{\}\tau\{B, C\}[B = d, C = d]\{\}\tau\{B, C\}[B = d, C = d]\cdots.$$

According to the definition of a synchronous channel in Reo, the channel coordinates the data exchange between the ports to be simultaneous. The ABAR model illustrated in Figure 5.4(c) more explicitly shows the semantics for Reo's Sync channel than the ABAR model illustrated in Figure 5.4(b). It is easy to see that the two automata are visibly equivalent. In a similar way, the synchronous channel can more explicitly be modeled by considering two other passible states, one with the label $B.\bar{C}$ and the other with the label $\bar{B}.C$.

Other basic Reo connectors can also be modeled by ABARs:

A synchronous drain (and similarly for the synchronous spout) between two ports B and C can be modeled as a synchronous channel, but for the data values passing through the two ports that in this case needs not to be the same:



,

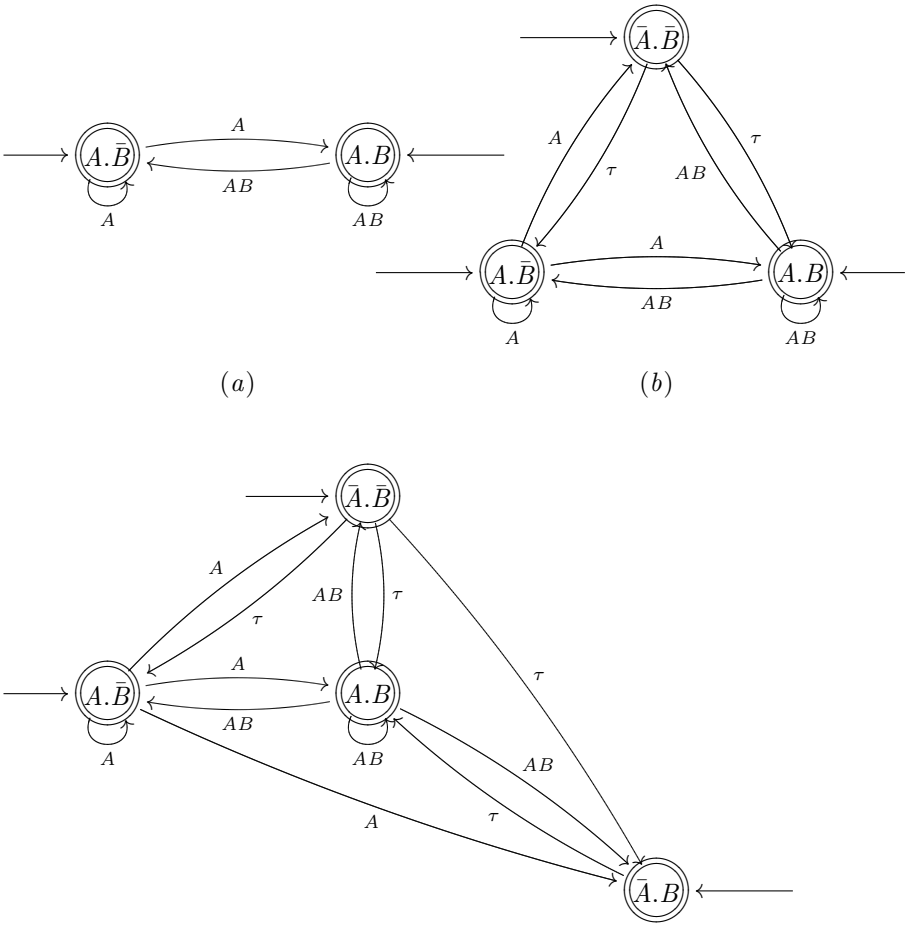


Figure 5.2: Three ABAR models of the context dependent lossy synchronous channel

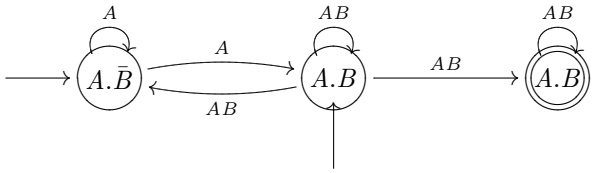


Figure 5.3: The ABAR model of a fair closed system of a context dependent lossy synchronous channel and its environment

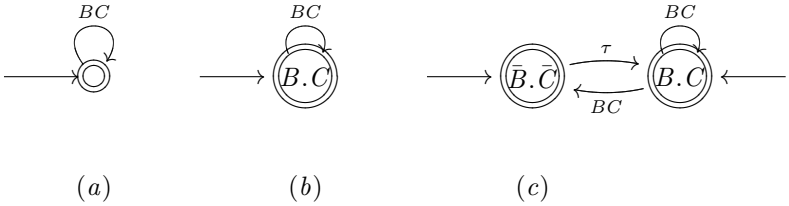
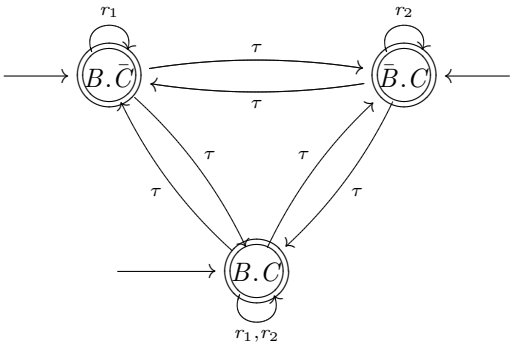


Figure 5.4: Models for a Reo synchronous channel (Sync) from source node B to sink C: (a) Its BAR model; (b) The canonical ABAR model for (a); and (c) The more explicit ABAR model.

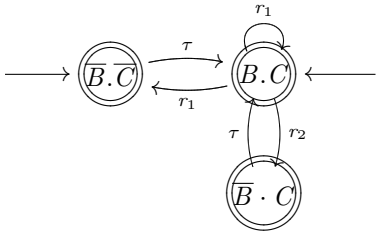
where $dom(r) = \{B, C\}$. Note that based on our definition of context dependency (presented in Section 5.1) synchronous drain and synchronous spout channels are not context dependent.

The asynchronous version of a drain channel between B and C can be modeled by the following ABAR:



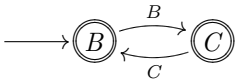
where $dom(r_1) = \{B\}$ and $dom(r_2) = \{C\}$. Note that this channel is (non-trivially) nondeterministic: when write requests exists on both of its ports, the channel can choose to consume either one of them. Thus, in the case of this channel, we can consider some fairness conditions, such as, the requirement that the input data on each port should be consumed infinitely often. Obviously, we can model this fair asynchronous drain by an ABAR with more states not all of which are accepting.

A filter channel from B to C is a synchronous channel that allows for the communication of data items that have a special value. We can model this pattern using the record $[B = p, C = p]$ where p is the special value of the filter. Thus, the ABAR model of filter channel is:



where $r_1 \doteq [B = p, C = p]$, $dom(r_2) = \{B\}$, and $r_2.B \neq p$.

Finally, a FIFO1 channel from B to C is an asynchronous channel that has a buffer with capacity one. Thus, the ABAR model of a FIFO1 channel over a singleton data set is:



5.4 Composing ABAR Models

Now, we introduce the counterpart composition operators that we introduced for BAR's in the case of ABAR's. Again we show that the join operation can be split into two more basic operations: name extension and product.

5.4.1 Product and Join

In this section we give a definition of product and join of two ABAR's.

Definition 5.10 Let $\langle B_1, l_1\rangle$ and $\langle B_2, l_2\rangle$ be two ABAR over the same alphabet, say $Rec_{\mathcal{N}}(\mathcal{D})$. Their *product* is defined as the ABAR $\langle B, l\rangle$, where $B = B_1 \times B_2$ and $l(\langle q, p\rangle) = l_1(q).l_2(p)$.

Similarly, we define the join of two ABARs in terms of the join of their underlying BAR's.

Definition 5.11 Let $\langle B_1, l_1\rangle$ and $\langle B_2, l_2\rangle$ be two ABAR over the same alphabet, say $Rec_{\mathcal{N}}(\mathcal{D})$. Their *join* $\langle B_1, l_1\rangle \bowtie \langle B_2, l_2\rangle$ is defined as the ABAR $\langle B, l\rangle$, where $B = B_1 \bowtie B_2$ and $l(\langle q, p\rangle) = l_1(q).l_2(p)$.

It is easy to check that the join of ABARs is again an ABAR. In fact, if $q_1 \xrightarrow{r_1} p_1$ is a transition in $\langle B_1, l_1\rangle$ and $dom(r_1)$ has no name in common with those used by another ABAR $\langle B_2, l_2\rangle$, then $l_1(q_1).l_2(q_2)$ implies $wp(r_1)$ for all state q_2 of B_2. Similarly, if $q_2 \xrightarrow{r_2} p_2$ is another transition in $\langle B_2, l_2\rangle$ such that $comp(r_1, r_2)$, then $l_1(q_1).l_2(q_2)$ implies $wp(r_1 \cup r_2)$.

As for BAR's, the join of two ABAR's with the same alphabet coincides with their product. In general, the join operator is not a congruence with respect to the visible equivalence.

To see this, it is enough to take two visibly equivalent ABARs with one state labeled in one automaton with $A.B$ and in the other automaton by $A.\bar{B}$. The join of one of them with an automata with a state labeled by B is different than the join of the other.

We now give an example of connector composition.

Example 5.4 Consider the context dependent lossy synchronous channel from port A to port B given in Figures 5.2(a), (b) and (c) and the synchronous channel from B to C as modeled in Figure 5.4(c). Their joint automata are respectively the ABAR models shown in Figures 5.5(a), (b) and (c). Note that they are very similar to their corresponding models of the context dependent lossy synchronous channel between port A and C, except that we can still observe the data-flow through the port B. After hiding port B, each automaton will be the same as its corresponding context dependent lossy synchronous channel.

5.4.2 Hiding of Port Names

Now, we define the hiding operator for the case of ABAR's as the counterpart of the hiding operator we previously defined for BAR's.

Definition 5.12 Let $\langle B, l\rangle$ be an ABAR. The hiding of a port A results in the ABAR $\langle B{\downarrow_A}, l'\rangle$ where $l'(q)$ is the expression $l(q)[1/\bar{A}][1/A]$ which means that we first substitute 1 for every occurrence of $\bar{A}$ and then substitute 1 for every occurrence of A in $l(q)$.

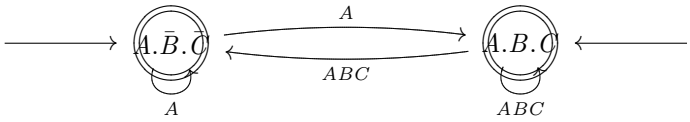
For example, using the above definition, if we hide port B in all three ABAR models illustrated in Figure 5.5 we obtain exactly the ABAR models of a lossy synchronous channel from A to C as illustrated in Figure 5.2 (after a renaming of the sink port which now is C not B).

Now consider some more complex examples of joining of context dependent connectors and then hiding the common ports:

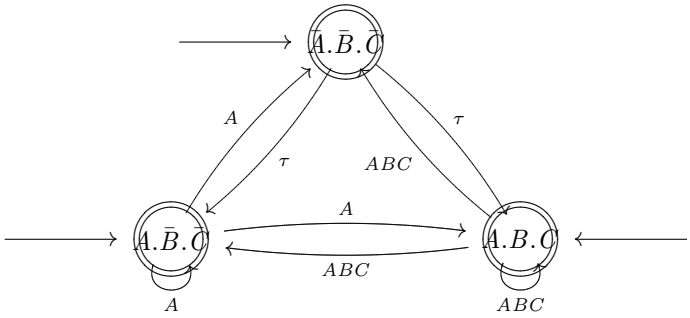
Example 5.5 In Figure 5.6 we consider the composition of a context dependent lossy synchronous channel from port A to B with another one from port B to C. The resulting ABARs before and after hiding the common port B are illustrated in Figures 5.6(c) and (d). As we expect, in the product automaton (before hiding), the first channel (from port A to B) indeed always acts as a normal synchronous channel; i.e., it never loses. The resulting connector is exactly an ABAR model of a context dependent lossy synchronous channel from A to C.

Example 5.6 In Figure 5.7 we consider the composition of a context dependent lossy synchronous channel from port A to B with a FIFO1 channel from port B to C, after hiding the common port B. Note that in the initial state of the resulting connector the buffer is empty and in the two other states the buffer is full. As we expect, whenever the buffer is empty no data value from port A is lost, whereas this happen when the buffer is full.

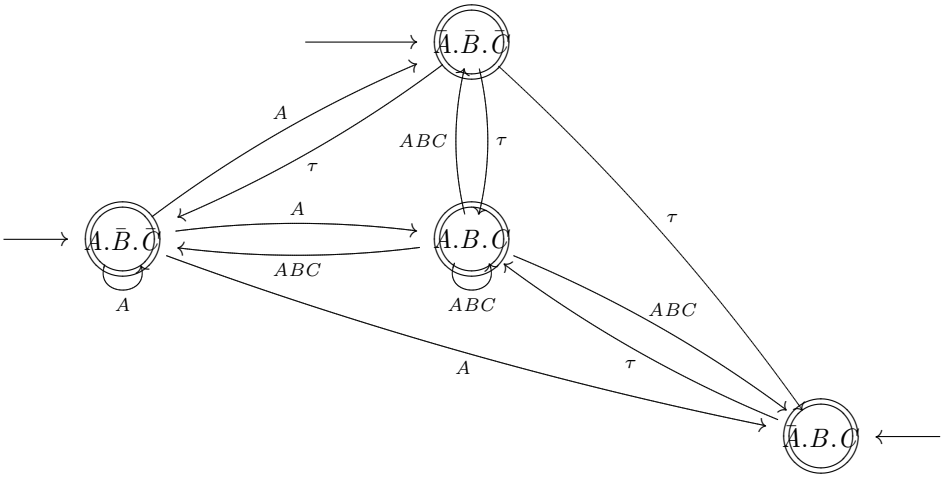
Example 5.7 Consider the composition of context dependent lossy synchronous and FIFO1 channels in a reverse direction as we did in Example 5.6. In Figure 5.8 we consider the composition of a FIFO1 channel from port A to B with a context dependent lossy synchronous channel from port B to C, after hiding the common port B. In the two initial states the buffer is empty and in the others it is full. As we expect, when the buffer is empty firing port A



(a)



(b)



(c)

Figure 5.5: The composition of the ABAR models of a context dependent lossy synchronous channel and a synchronous channel

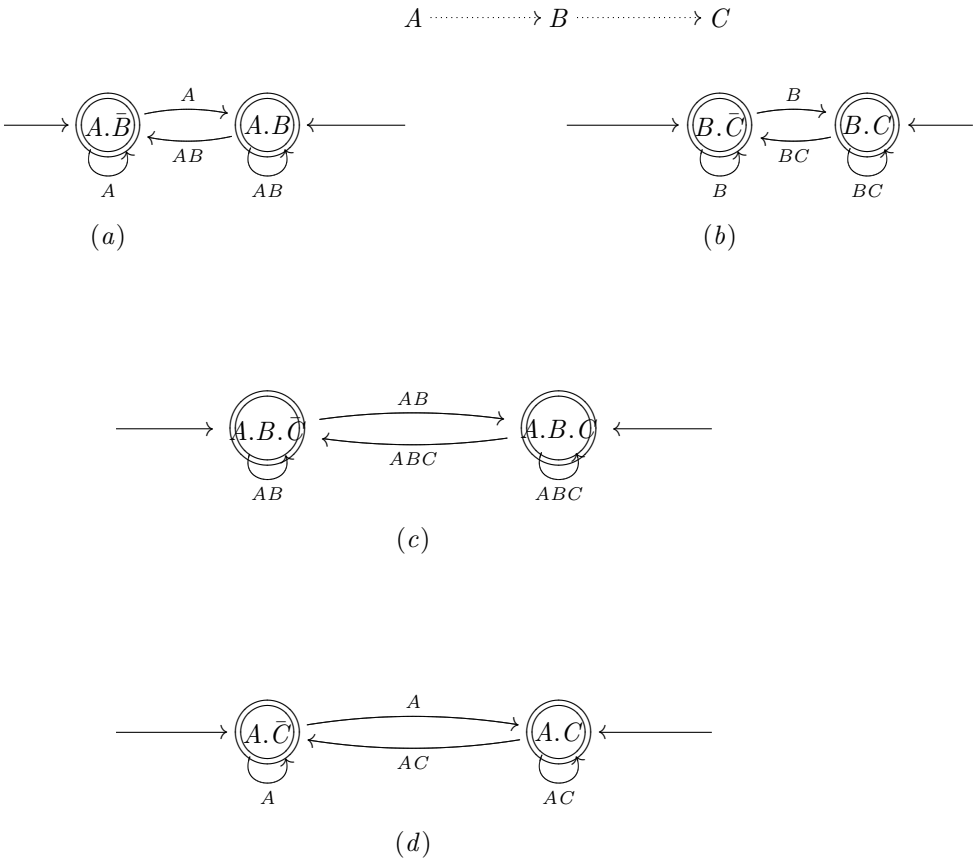


Figure 5.6: The composition of two context dependent lossy synchronous channels.

causes the buffer to become full. In the next stage, if C is ready to get data, it receives it, but if C is not ready data will be lost and either way the buffer becomes empty.

Example 5.8 In Figure 5.9 we consider the composition of a synchronous channel from port A to B with a FIFO1 channel from port B to C, after hiding the common port B. As we expect, the obtained ABAR is visibly equivalent to a FIFO1 channel from source port A to sink port C.

Example 5.9 Consider the composition of a synchronous and a FIFO1 channel in a reverse direction as we did in Example 5.8. In Figure 5.10 we consider the composition of a FIFO1 channel from port A to B with a synchronous channel from port B to C, after hiding the common port B. As we expect the obtained model is a FIFO1 channel from A to C.

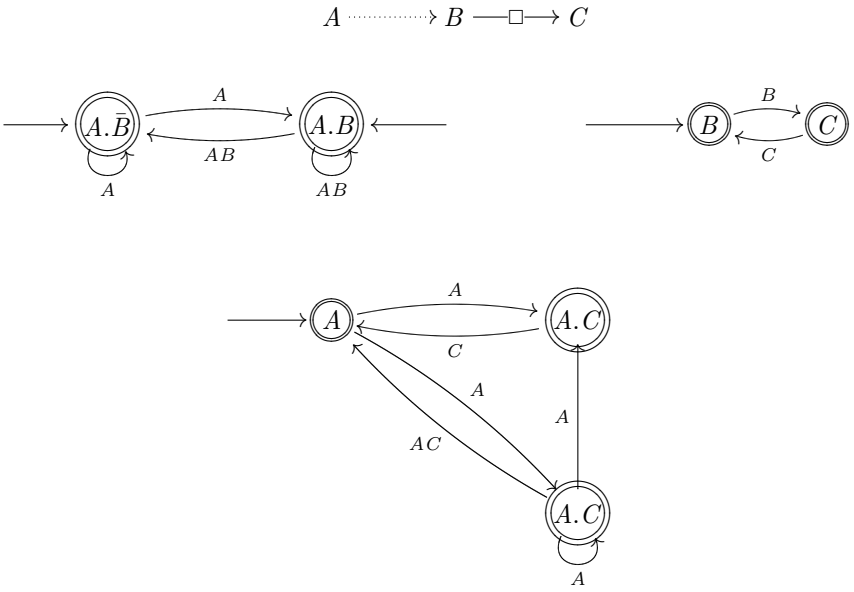


Figure 5.7: The composition of a context dependent lossy synchronous channel with a FIFO1 channel.

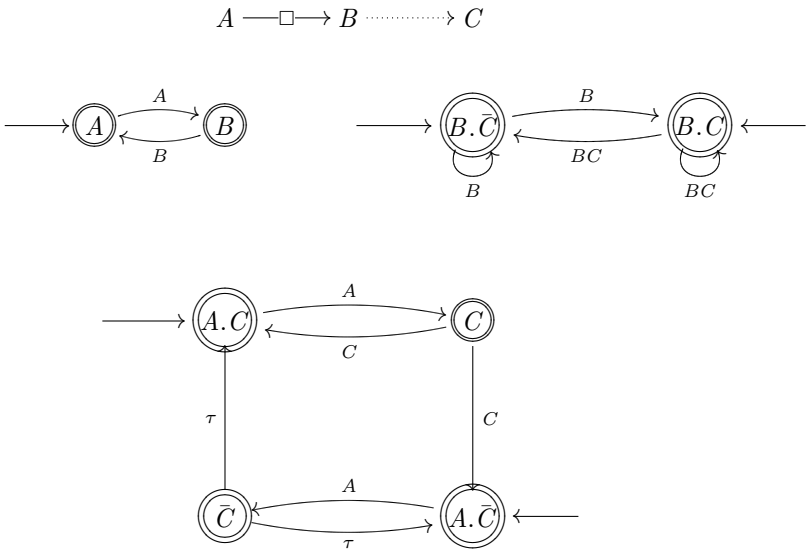


Figure 5.8: The composition of a FIFO1 channel with a context dependent lossy synchronous channel.

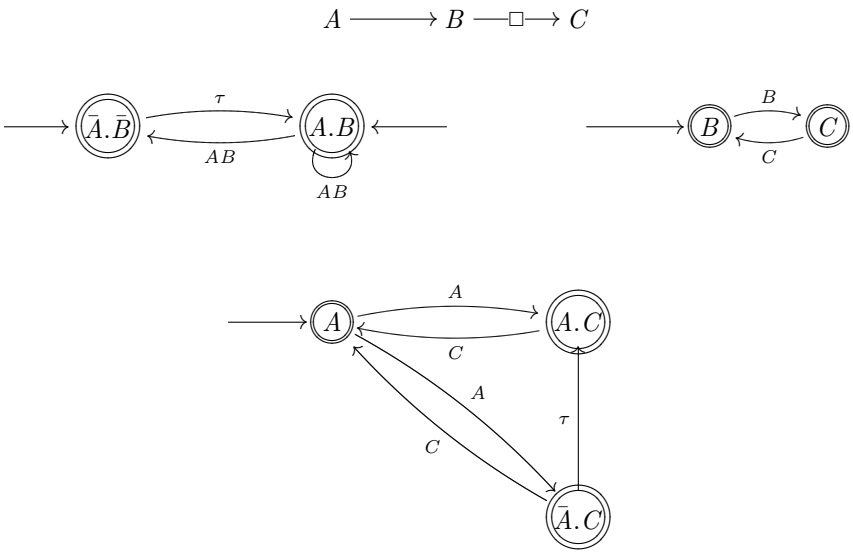


Figure 5.9: The composition of a synchronous channel with a FIFO1 channel.

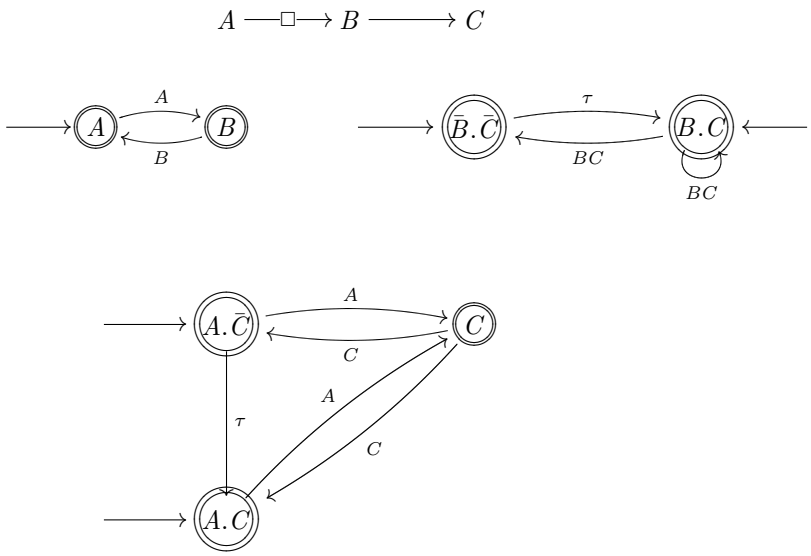


Figure 5.10: The composition of a FIFO1 with a synchronous channel.

5.4.3 Splitting the join

We now show that the procedure of splitting the join into name extension and production that we introduced for BARs is applicable to ABARs as well.

Theorem 5.1 Let $\langle B_1, l_1\rangle$ and $\langle B_2, l_2\rangle$ be two ABARs over the alphabet sets $Rec_{\mathcal{N}_1}(\mathcal{D})$ and $Rec_{\mathcal{N}_2}(\mathcal{D})$, respectively. Then,

$$\langle B_1{\uparrow}\mathcal{N}_2 \times B_2{\uparrow}\mathcal{N}_1, l'\rangle = \langle B_1, l_1\rangle \bowtie \langle B_2, l_2\rangle,$$

where $l'(\langle q_1, q_2\rangle) = l_1(q_1).l_2(q_2)$, and q_1 is a state of B_1 and q_2 is a state of B_2.

Proof. The proof is a simple extension of the proof of Theorem 4.7. □

Example 5.10 Consider Figure 5.11. Figures 5.11(a) and 5.11(b) show the simplest ABAR models of two FIFO1 channels (over a singleton data set $\mathcal{D} = \{d\}$). They are the same BAR models we presented in the previous chapter, now augmented with proper labels. The extension of the first ABAR with port name C appears in Figure 5.11(d), while the extension of the second automaton with port name A appears in Figure 5.11(e). Their product is the automaton in 5.11(c) which is obtainable using either the direct or the splitting definitions of the join operation.

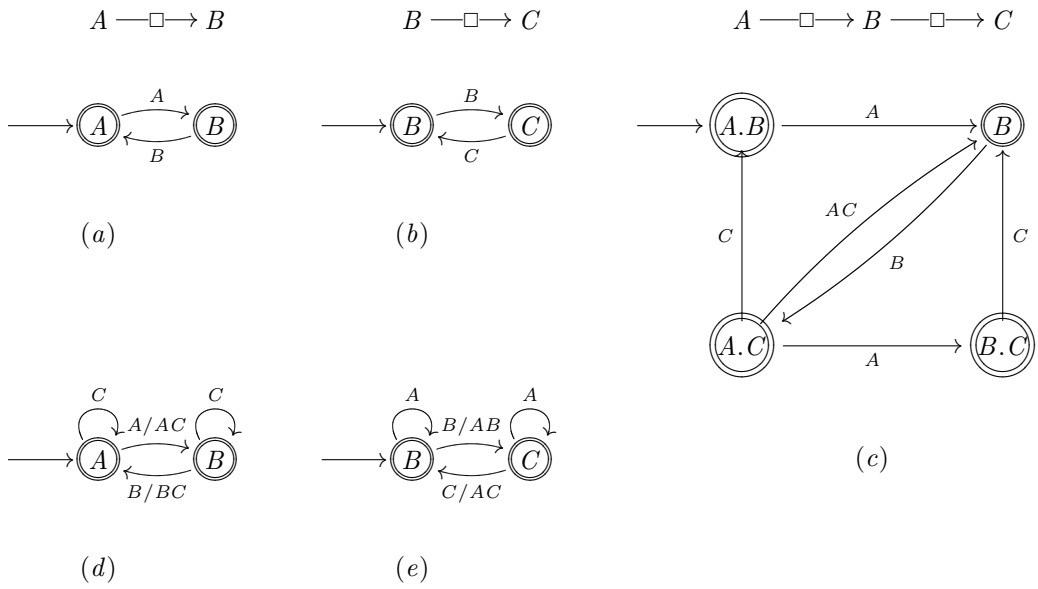


Figure 5.11: Direct and indirect joining of two FIFO1 buffers modeled by ABARs

5.5 Context Dependent Fair Constraint Automata

In Section 4.6 we introduced a new notion and semantics for constraint automaton called *fair constraint automaton* (FCA). The syntax of FCA is the same as that of ordinary constraint automaton except that it also has some final states. From the semantics point of view, each FCA is defined as an acceptor of infinite strings of records. An infinite string of records is accepted by an FCA if at least one of the accepting states occurs in the execution for the string infinitely many times. Now, we investigate the augmentation of FCAs by test expression labels as we did in this chapter for BARs. Let $C = \langle Q, \mathcal{N}, \longrightarrow, Q_0, F \rangle$ be a FCA over a port set $\mathcal{N}$ and a data set $\mathcal{D}$, as in Definition 4.14. Also, the set of all test expressions over the port set $\mathcal{N}$ and the data set $\mathcal{D}$ be the same as in Definition 5.1, with their semantics as in Definition 5.2. Now we define the augmentation of FCAs:

Definition 5.13 An augmented fair constrain automaton (abbreviated by AFCA) is a pair $\langle C, l \rangle$ consisting of an FCA $C = \langle Q, \mathcal{N}, \longrightarrow, Q_0, F \rangle$ and a labeling function $l : Q \to Exp_{\mathcal{N}}$ such that for all $q \in Q$, if $q \xrightarrow{N,q} q'$ then $l(q)$ implies $wp(N)$, where for every $N \subseteq \mathcal{N}$, $wp(N)$ is defined inductively as follows:

$$wp(\emptyset) = 1$$
$$wp(N) = A \cdot wp(N \setminus A) \quad \text{if } A \in N.$$

As a consequence of the above definition, if $l(q) = \bar{A}$, then all transitions outgoing from

q must be internal, i.e., they must be labelled by τ. Similarly, all transitions outgoing from a state labeled by 1 must be internal.

Same as the case of ABARs, we regard AFCAs as acceptors of infinite guarded strings of records:

Definition 5.14
(1) Let $\gamma = M_0 r_0 M_1 r_1 \cdots$ be an infinite guarded string over the alphabet $Rec_{\mathcal{N}}(\mathcal{D})$. We define an *infinite computation* for γ in an AFCA $\langle C, l\rangle$ (over the same sets $\mathcal{N}$ and $\mathcal{D}$) to be an infinite sequence $\pi = q_0, (N_0, g_0), q_1, (N_1, g_1), ...$, of alternating states and (port set , guard) pairs in which $q_0 \in Q_0$, $M_i \models l(q_i)$, and there is data assignment $\delta{:}N_i \rightarrow \mathcal{D}$ such that $\delta \models g_i$, $dom(r_i) = N_i$ and $\forall n \in N_i{:}\, r.n = \delta(n)$.
(2) An infinite guarded strings γ is *accepted* by AFCA $\langle C, l\rangle$ if there is an infinite computation for γ in $\langle C, l\rangle$ with at least one of the final states occurring infinitely often.
(3) The *guarded language* of an AFCA $\langle C, l\rangle$, denoted by $GL(C)$, is the set of all infinite guarded strings accepted by it.

Definition 5.15
(1) Two AFCAs C_1 and C_2 are *guarded-language equivalent* if $GL(C_1) = GL(C_2)$.
(2) AFCA C and BAR C' are *(language) equivalent* if after ignoring the labeling function of C and considering its language of infinite strings of record we have $L(C) = L(C')$.

Given an AFCA $\langle C, l\rangle$ we can construct a guarded-language equivalent AFCA $\langle C', l'\rangle$ such that $l'(q) \not\equiv 0$ for all states q of B'. In fact, we can safely delete these inconsistent states from the set of states of B and their adjunct transitions because no set of names N will ever pass the test 0 (not even the empty set of names).

An augmented fair constraint automaton (AFCA) can be considered as a fair constraint automaton (FCA), if we ignore its labeling function.

Conversely, every fair constraint automaton C can be transformed into a canonical AFCA $\langle C, l\rangle$ by assigning to each state q of C the conjunction of all $wp(N)$ for each $N \in \mathcal{N}$ that is the first component of a pair (N, g) labeling the outgoing transitions from q. Namely:

Definition 5.16 Let $C = \langle Q, \mathcal{N}, \longrightarrow, Q_0, F\rangle$ be an FCA over a port set $\mathcal{N}$ and a data set $\mathcal{D}$. The *canonical AFCA* for C is the AFCA $\langle C, l\rangle$ where the labeling function is define as follows:

$$\forall q \in Q, l(q) = \bigwedge_{N \in W} wp(N)$$

and

$$W = \{N \mid \exists q \xrightarrow{N,g} q' \in C\}.$$

Obviously, if C is an FCA and C' is its canonical AFCA then considering their languages of streams of records, they are equivalent. Transforming a BAR into its canonical ABAR and back will produce the same BAR, while the converse holds only for an ABAR without states with negative tests.

Let us compare the expressiveness of ABARs and AFCAs. Obviously, because the semantics of both augmented Büchi automata of records and augmented fair constraint automata are based on guarded languages of streams of records, each ABAR B over a name set $\mathcal{N}$ and a data set $\mathcal{D}$ can be considered as an AFCA if we replace every transition label $r \in Rec_{\mathcal{N}}(\mathcal{D})$

with (N, g) where, $N = dom(r)$ and g is the data constraint $\bigwedge_{n \in dom(n)}(d_n = r.n)$. Using this simple conversion of ABAR into AFCA, we can show that all ABARs that we introduced as models of Reo connectors can be considered as AFCA models of them.

Conversely, if C is an AFCA over a finite name set $\mathcal{N}$ and a finite data set $\mathcal{D}$ then C is equivalent with ABAR B all whose components are the same as C except that each transition $q \xrightarrow{N,g} q'$ of C is replaced with a set of transitions of the form $q \xrightarrow{r} q'$ where r satisfies the following conditions: (1) $dom(r) = N$, and (2) there exists a data assignment $\delta{:}N \to \mathcal{D}$ such that $\delta \models g$ and $\forall n \in N$, $\delta(n) = r.n$. Obviously, if at least one of the sets $\mathcal{N}$ or $\mathcal{D}$ is infinite then in replacing an AFCA's transitions with a set of transitions with record labels, we will need to have records with infinite domains or to replace the transition of the AFCA with an infinite set of transitions with record labels. This density in the syntax of AFCAs in comparison with the syntax of ABAR's is the main advantage of using AFCA instead of ABAR.

6 Model Checking

In the previous chapters, we introduced Büchi automata of records and their augmented versions as the operational models of Reo connectors considering unconditional fairness and context dependency requirements. Now we have an operational semantics for Reo based on Büchi automata. Thus, it is very natural to use these models for automata theoretic model checking of Reo nets. Generally speaking, *automata theoretic methods of model checking* verify if a system satisfies a desired property using three steps. First, we model the system by an automaton and specify the property by a formula in a temporal logic. The next steps are applicable if there is a well established translation of the formulas of the selected temporal logic into the selected type of automata. In this case, the second step is to translate the negation of the formula that expresses the desired property into an automaton of the same type as the one used to model the system. Now, we have two automata of the same type. The last step is to check the intersection of their languages. If the intersection is empty, it will be a proof that the system satisfies the property. Otherwise, each member of the intersection set is a counterexample trace of the system that violates the property. The model checking process can be improved (both from time and space complexity points of view) if we can do the generation of the state space of the automaton model of the system or the translation of the property formula into its equivalent automaton and the checking of the emptiness of their intersection in parallel. If some form of this parallelism is possible, we call the model checking process *on-the-fly*.

In this chapter, we try to use the automata theoretic method of model checking for systems modeled by ABARs. First, we introduce an action (or transition) based linear temporal logic (called ρLTL) interpreted over computations of ABARs. Then, we show that ρLTL formulas can be translated into ABARs both using inductive and on-the-fly methods. In each case, we obtain a technique to verify Reo nets.

6.1 Record-based linear-time temporal logic

In this section we introduce a record based linear time temporal logic (ρLTL) which is an extension of linear time temporal logic (LTL) [145] for reasoning about data-flow, synchronization and context dependencies of Reo connectors. We use as atomic propositions sets of port names, indicating the ports ready to communicate, and index the usual next state operator of LTL with a record, for the specification of communicating ports and of their respective data-flow.

Definition 6.1 Syntax of ρLTL. The set of *ρLTL formulas* over a finite set of port names $\mathcal{N}$ and a finite set of data $\mathcal{D}$ is defined inductively by the following syntax:

$$\phi ::= \ true \mid N \mid \neg\phi \mid \phi \vee \phi \mid \langle r \rangle \phi \mid \phi \, U \, \phi.$$

where $N \subseteq \mathcal{N}$ and $r \in Rec_{\mathcal{N}}(\mathcal{D})$.

Formulas of ρLTL are interpreted over infinite guarded strings. A necessary condition to interpret a formula for a guarded string is that both use the same set of port names $\mathcal{N}$ and data set $\mathcal{D}$, which we assume to hold in the sequel. Intuitively, N holds for a guarded string

if N is the first *guard* of the string, whereas $\langle r \rangle \phi$ holds if r is the first *action* of the string and ϕ holds for its remaining suffix.

Formally, given an infinite guarded string $M = N_0 r_0 N_1 r_1 \cdots$, we define M^i as the guarded string $N_i r_i N_{i+1} r_{i+1} \cdots$. Here we consider only guarded strings for which $N_i \subseteq \mathcal{N}$ and $r_i \in Rec_{\mathcal{N}}(\mathcal{D})$, for all $i \geq 0$.

Definition 6.2 Semantics of ρLTL. Let $M = N_0 r_0 N_1 r_1 \cdots$ be an infinite guarded string over a name set $\mathcal{N}$ and a data domain $\mathcal{D}$ such that $\forall i \geq 0, dom(r_i) \subseteq N_i$. The semantics of a ρLTL formula is defined inductively over such M's as follows:

$$
\begin{array}{lll}
M \models true & & \\
M \models N & \text{iff} & N_0 = N \\
M \models \phi_1 \vee \phi_2 & \text{iff} & M \models \phi_1 \text{ or } M \models \phi_2 \\
M \models \neg\phi & \text{iff} & M \not\models \phi \\
M \models \langle r \rangle \phi & \text{iff} & r_0 = r \text{ and } M^1 \models \phi \\
M \models \phi_1 U \phi_2 & \text{iff} & \exists j \geq 0 \text{ such that } M^j \models \phi_2 \text{ and } \forall 0 \leq i < j, M^i \models \phi_1
\end{array}
$$

Based on the above semantic definitions and the intuitions behind them, the temporal operators $\langle r \rangle$ and U are called *(action-based) next* and *until* operators respectively. As usual, we denote by $\| \phi \|$ the set of all models of the ρLTL formula ϕ, and define logical equivalence $\equiv$ of ρLTL formulas as $\phi_1 \equiv \phi_2$ if and only if $\| \phi_1 \| = \| \phi_2 \|$. If B is an ABAR and ϕ a ρLTL formula, we write $B \models \phi$ if $L(B) \subseteq \| \phi \|$.

Several other operators can be derived from the basic operators of ρLTL. $false$ is defined as $\neg true$. The Boolean operators $\wedge$ and $\rightarrow$ are derived in the obvious way:

$$\phi_1 \wedge \phi_2 = \neg(\neg\phi_1 \vee \neg\phi_2),$$

$$\phi_1 \rightarrow \phi_2 = \neg(\phi_1 \vee \neg\phi_2).$$

The temporal modalities *eventually* and *always* can be derived as usual:

$$\Diamond\phi = true U \phi,$$

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

The dual operator of *until* is the *release* operator defined as:

$$\phi \, R \, \psi = \neg(\neg\phi \; U \; \neg\psi).$$

The *weak* variant 'W' of the until operator is obtained as:

$$\phi \; W \; \psi = (\phi \; U \; \psi) \vee \Box\phi.$$

Using the fact that if $M = N_0 r_0 N_1 r_1 \cdots$ is a guarded string that is used as the semantic domain of ρLTL formulas then $\forall i \geq 0, dom(r_i) \subseteq N_i$ and that $\mathcal{N}$ is finite, we can conclude hat:

$$\langle r \rangle \phi \equiv \Big(\bigvee_{\mathcal{N} \supseteq N \supseteq dom(r)} N \Big) \wedge \langle r \rangle \phi.$$

Based on the above fact, we can also derive other nice equivalences such as this:

$$\{A\} \wedge \langle [A = 1, B = 1] \rangle true \ \equiv \ false.$$

The dual operator of $\langle r \rangle \phi$ is

$$[r]\phi = \neg \langle r \rangle \neg \phi$$

which intuitively holds for a guarded string if either its first action is other than r or its continuation satisfies ϕ. In fact, $[r]\phi \equiv \neg \langle r \rangle true \vee \langle r \rangle \phi$. For example, the formula $[r]false$ is satisfied by all guarded strings having a record other than r as their first action. We prove this equivalence in the following lemma:

Lemma 6.1

$$[r]\phi \equiv \neg \langle r \rangle true \vee \langle r \rangle \phi.$$

Proof. Let $M = N_0 r_0 N_1 r_1 \cdots$ be a guarded string. Now,

$$\begin{array}{lll} M \models [r]\phi & \text{iff} & M \models \neg \langle r \rangle \neg \phi \\ & \text{iff} & M \not\models \langle r \rangle \neg \phi \\ & \text{iff} & \text{it is not the case that } (r_0 = r \text{ and } M^1 \models \neg \phi) \\ & \text{iff} & r_0 \neq r \text{ or } M^1 \models \phi \\ & \text{iff} & M \models \neg \langle r \rangle true \text{ or } M \models \langle r \rangle \phi \\ & \text{iff} & M \models \neg \langle r \rangle true \vee \langle r \rangle \phi. \end{array}$$

□

6.1.1 Some useful encodings

Because there are only finitely many records in $Rec_{\mathcal{N}}(\mathcal{D})$, the standard *next* operator of linear time temporal logic can be defined as:

$$\bigcirc \phi = \bigvee_{r \in Rec_{\mathcal{N}}(\mathcal{D})} \langle r \rangle \phi.$$

It is not hard to see that the next operator is self-dual, in the sense that

$$\neg \bigcirc \phi \equiv \bigcirc \neg \phi.$$

Further, because our models are infinite strings, $\bigcirc true \equiv true$, meaning that connectors are reactive and cannot stop the data flow (progress is always possible).

Two important equivalences are the definitions of *until* (U) and *release* (R) temporal operators using a recursive style [29]:

$$\phi_1 \ U \ \phi_2 \ \equiv \ \phi_2 \vee (\phi_1 \wedge \bigcirc(\phi_1 \ U \ \phi_2)),$$

$$\phi_1 \ R \ \phi_2 \ \equiv \ \phi_2 \wedge (\phi_1 \vee \bigcirc(\phi_1 \ R \ \phi_2)).$$

Definition 6.3 A *data constraint* δ for a set of names $N \subseteq \mathcal{N}$ is a satisfiable propositional formula built from the atoms $d_A \in P$, $d_A = d$, and $d_A = d_B$, where $A, B \in N$, $d \in \mathcal{D}$ and $P \subseteq \mathcal{D}$.

Data constraints, together with a set of names on which they are defined can be viewed as a symbolic representation of a set of records. We can therefore define a derived operator $\langle N, \delta\rangle\phi$, where δ is a data constraint for N, by setting

$$\langle N, \delta\rangle\phi = \bigvee\{\langle r\rangle\phi \mid dom(r) = N, r \models \delta\}\,,$$

where $r \models (d_A \in P)$ if $r.A \in P$, $r \models (d_A = d)$ if $r.A = d$ and $r \models (d_A = d_B)$ if $r.A = r.B$ (disjunction and negation are as expected).

In [17, 18], timed scheduled-data expressions are introduced to specify data stream logic. Leaving out time, *scheduled-data expressions* are ordinary regular expressions built from either data constraints or records. Scheduled-data expressions α are incorporated in data stream logic by formulas of the type $\langle\langle\alpha\rangle\rangle\phi$. More formally, a scheduled-data expression α can be defined using the following abstract grammar:

$$\alpha ::= 0 \mid 1 \mid \langle N, \delta\rangle \mid r \mid \alpha + \alpha \mid \alpha \times \alpha \mid \alpha;\alpha \mid \alpha^*.$$

While in general standard linear temporal logic cannot express regular expressions for prefixes of infinite strings, we can encode scheduled-data expressions in our action based linear temporal logic ρLTL by using a function $(-)^\dagger$ that maps scheduled-data expressions of the form $\langle\langle\alpha\rangle\rangle\phi$ into ρLTL formulas. The function $(-)^\dagger$ is defined recursively as follows:

$$\begin{array}{lcl}
(true)^\dagger & = & true \\
(N)^\dagger & = & N \\
(\langle\langle 0\rangle\rangle\phi)^\dagger & = & false \\
(\langle\langle 1\rangle\rangle\phi)^\dagger & = & (\phi)^\dagger \\
(\langle\langle \langle N, \delta\rangle\rangle\phi)^\dagger & = & \langle N, \delta\rangle(\phi)^\dagger \\
(\langle\langle \langle r\rangle\rangle\phi)^\dagger & = & \langle r\rangle(\phi)^\dagger \\
(\langle\langle \alpha_1 + \alpha_2\rangle\rangle\phi)^\dagger & = & (\langle\langle \alpha_1\rangle\rangle\phi)^\dagger \vee (\langle\langle \alpha_2\rangle\rangle\phi)^\dagger \\
(\langle\langle \alpha_1 \times \alpha_2\rangle\rangle\phi)^\dagger & = & (\langle\langle \alpha_1\rangle\rangle\phi)^\dagger \wedge (\langle\langle \alpha_2\rangle\rangle\phi)^\dagger \\
(\langle\langle \alpha_1;\alpha_2\rangle\rangle\phi)^\dagger & = & (\langle\langle \alpha_1\rangle\rangle(\langle\langle \alpha_2\rangle\rangle\phi))^\dagger \\
(\langle\langle \alpha^*\rangle\rangle\phi)^\dagger & = & (\langle\langle \alpha\rangle\rangle true)^\dagger\, U (\phi)^\dagger \\
(\phi \wedge \psi)^\dagger & = & (\phi)^\dagger \wedge (\psi)^\dagger
\end{array}$$

Note that 0 is the unit with respect to the union operator $+$, and 1 is the unit with respect to the composition operator ;. In fact we have

$$\begin{array}{l}
\langle\langle 0 + \alpha\rangle\rangle\phi \equiv \langle\langle 0\rangle\rangle\phi \vee \langle\langle \alpha\rangle\rangle\phi \equiv false \vee \langle\langle \alpha\rangle\rangle\phi \equiv \langle\langle \alpha\rangle\rangle\phi \\
\langle\langle 1;\alpha\rangle\rangle\phi \equiv \langle\langle 1\rangle\rangle(\langle\langle \alpha\rangle\rangle\phi) \equiv \langle\langle \alpha\rangle\rangle\phi\,.
\end{array}$$

Scheduled-data expressions allow us to express formulas that hold only for externally observable steps, thus not sensible for a finite number of internal steps. Given a ρLTL formula ϕ, we define $\Diamond_\tau\phi = \langle\langle\tau^*\rangle\rangle([\tau]false \wedge \phi)$. Informally, $\Diamond_\tau\phi$ holds if ϕ holds after finitely many internal τ steps.

6.1.2 Specifying Reo connectors

We now present some examples of specification of basic Reo connectors using ρLTL formulas. First, for simplicity, the ABARs depicted in Figure 6.1 are considered as models of some

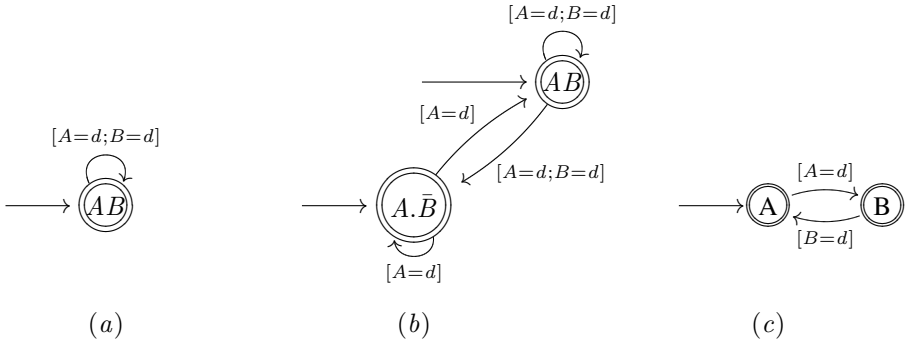


Figure 6.1: ABAR models of some basic Reo connectors: (a) Sync channel, (b) Context-Dependent LossySync channel, and (c) FIFO1 channel.

of the basic Reo connectors between two ports A and B over a singleton data set $\{d\}$. The labels of the states are sets of port names that are ready before firing the outgoing transitions.

Now, consider a synchronous channel from a port A to a port B. If both ports are enabled, then the channel must let the data flow. This can be expressed by the following ρLTL formula:

$$\phi \;=\; \Box(\{A, B\} \rightarrow \langle\langle\{A, B\}, d_A = d_B\rangle\rangle true)\,. \tag{6.1}$$

The above formula is clearly satisfied by our ABAR model of synchronous channel in Figure 6.1. However, it is also satisfied by the ABAR modeling a lossy synchronous channel. This is because (6.1) does not guarantee that data cannot flow through a single port. We remedy this by adding to the specification of a synchronous channel the following

$$\phi_1 \;=\; \Box(\neg\langle\langle\{A\}, true\rangle\rangle true \vee \neg\langle\langle\{B\}, true\rangle\rangle true)\,.$$

The above formula does not hold for the lossy synchronous channel. In fact, for such a connector it holds that if port A is enabled but B is not, then the data at A is lost. This is expressed by

$$\phi_2 \;=\; \Box((\{A\} \wedge \neg\{A, B\}) \rightarrow \langle\langle\{A\}, true\rangle\rangle true)$$

Further, in a lossy synchronous channel, data cannot flow through port B alone, that is

$$\phi_3 \;=\; \Box\neg\langle\langle B, true\rangle\rangle true.$$

Thus, a possible specification of the synchronous channel is

$$Sync = \phi \wedge \phi_1$$

while a specification of a lossy synchronous channel of Reo is

$$LossySync = \phi \wedge \phi_2 \wedge \phi_3.$$

Differen than the two previous channels, a FIFO1 channel is asynchronous, meaning that data does not flow simultaneously through its ports A and B, that is

$$\psi_1 \;=\; \Box\neg\langle\langle\{A, B\}, true\rangle\rangle true.$$

Further, a data item received through port A is never lost, as it is output to port B as soon as B is enabled. Of course, this does not need to be immediate and it can even be the case that B is never enabled. This is specified by means of a weak until operator allowing possibly infinitely many internal steps between the two observable actions:

$$\psi_2 \;=\; \Box \bigwedge_{d\in\mathcal{D}} \langle [A=d]\rangle(\langle\tau\rangle true \wedge \neg(\{B\}\vee\{A,B\}))\; W\; \langle [B=d]\rangle)\,.$$

To complete the specification of a FIFO1 channel, we need the converse of the above property, stating that after a data item flows through port B the store of the channel is empty and hence a new data item can flow through port A as soon as A is enabled:

$$\psi_3 \;=\; \Box\langle\langle\{B\}, true\rangle\rangle(\langle\tau\rangle true \wedge \neg(\{A\}\vee\{A,B\}))\; W\; \langle\langle\{A\}, true\rangle\rangle)\,.$$

Thus, in a FIFO1 channel, data flow through its two ports alternately, and never simultaneously. Summarizing, a specification for the FIFO1 channel is

$$FIFO1 \;=\; \psi_1 \wedge \psi_2 \wedge \psi_3.$$

6.2 From formulas to automata: model checking

In this section we introduce a global translation of ρLTL formulas into ABARs. Our construction is based on the translation from ordinary LTL formulas to Büchi automata [149], adapted to take into account the next state operator indexed by records. For simplicity, the resulting ABAR will have multiple sets of accepting states in which, a run is accepted if and only if for each accepting states set there exists at least one state that appears infinitely often in that run. Namely, we translate formulaes to generalized ABARs. To obtain an ordinary ABAR, one can use the fact that for each generalized Büchi automaton there is a language-equivalent ordinary Büchi automaton [138].

For technical convenience we will work with a positive form of ρLTL called ρLTL+.

Definition 6.4 Let $\mathcal{N}$ and $\mathcal{D}$ be respectively a finite nonempty set of port names and a finite nonempty set of data. The set of ρLTL+ formulas over sets $\mathcal{N}$ and $\mathcal{D}$ is the set of all formulas defined using the following abstract grammar:

$$\phi ::= \; true \,|\, false \,|\, N \,|\, \phi\wedge\phi \,|\, \phi\vee\phi \,|\; \bigcirc\phi \,|\, \langle r\rangle\phi \,|\, [r]\phi \,|\, \phi U\phi \,|\, \phi R\phi$$

where $N\subseteq\mathcal{N}$ and $r\in Rec_{\mathcal{N}}(\mathcal{D})$.

It is obvious that every ρLTL formula is equivalent to a positive one by pushing the negation inside every operator and replacing every instance of $\neg N$ with $\bigvee_{N'\subseteq\mathcal{N}, N'\neq N} N'$. Note that the size of the resulting positive formula is linear in the size of the ρLTL formula. The inclusion of the ordinary next state operator $\bigcirc\phi$ is to simplify the presentation.

We begin the translation of ρLTL+ formulas to automata by defining the closure $CL(\phi)$ of a ρLTL+ formula ϕ. Note that the closure mat include formulas that are not in the language of ρLTL+ (such as $\psi = \neg\langle r\rangle true$).

Definition 6.5 The closure $CL(\phi)$ of a ρLTL+ formula ϕ is the smallest set of ρLTL formulas such that:

- $\phi \in CL(\phi)$,
- $true, false \in CL(\phi)$,
- if there is $N \subseteq \mathcal{N}$ that $N \in CL(\phi)$ then for all $N' \subseteq \mathcal{N}$, $N' \in CL(\phi)$,
- if $\phi_1 \vee \phi_2 \in CL(\phi)$ then $\phi_1, \phi_2 \in CL(\phi)$,
- if $\phi_1 \wedge \phi_2 \in CL(\phi)$ then $\phi_1, \phi_2 \in CL(\phi)$,
- if $\bigcirc\psi \in CL(\phi)$ then $\psi \in CL(\phi)$ and for all $N' \subseteq \mathcal{N}$, $N' \in CL(\phi)$,
- if $\langle r\rangle\psi \in CL(\phi)$ then $\psi \in CL(\phi)$ and $dom(r) \in CL(\phi)$,
- if $[r]\psi \in CL(\phi)$ then $\neg\langle r\rangle true, \langle r\rangle\psi \in CL(\phi)$,
- if $\phi_1 \; U \; \phi_2 \in CL(\phi)$ then $\phi_1, \phi_2, \bigcirc(\phi_1 \; U \; \phi_2) \in CL(\phi)$,
- if $\phi_1 \; R \; \phi_2 \in CL(\phi)$ then $\phi_1, \phi_2, \bigcirc(\phi_1 \; R \; \phi_2) \in CL(\phi)$.

The set $CL(\phi)$ is finite, and its size is linear in the size of the formula ϕ.

The states of the ABAR associated with a formula ϕ are the propositionally and temporally consistent subsets of $CL(\phi)$, the so called *atoms*. Unlike the original Vardi-Wolper construction in [149] which allows only maximal consistent subsets, we allow any downward consistent subset of the closure to be an atom. Formally, we define atoms as follows:

Definition 6.6 An atom $A \subseteq CL(\phi)$ is a set such that

1. $true \in A$ and $false \notin A$,
2. for all $N \in CL(\phi)$, $N \in A$ if and only if for all $N' \neq N$, $N' \notin A$,
3. if $\phi_1 \vee \phi_2 \in A$ then $\phi_1 \in A$ or $\phi_2 \in A$,
4. if $\phi_1 \wedge \phi_2 \in A$ then $\phi_1 \in A$ and $\phi_2 \in A$,
5. if $\phi_1 \; U \; \phi_2 \in A$ then $\phi_2 \in A$ or $\phi_1, \bigcirc(\phi_1 \; U \; \phi_2) \in A$,
6. if $\phi_1 \; R \; \phi_2 \in A$ then $\phi_1, \phi_2 \in A$ or $\phi_2, \bigcirc(\phi_1 \; R \; \phi_2) \in A$,
7. if $[r]\psi \in A$ then $\neg\langle r\rangle true \in A$ or $\langle r\rangle\psi \in A$,
8. if $\langle r\rangle\psi \in A$ then there is $N \supseteq dom(r)$ such that $N \in A$,
9. if $\neg\langle r\rangle true \in A$ then there is $N \neq dom(r)$ such that $N \in A$,
10. if $\bigcirc\psi \in A$ then there is $N \subseteq \mathcal{N}$ that $N \in A$.

Now, we define the generalized ABAR counterpart of every ρLTL+ formula:

Definition 6.7 Let ϕ be a ρLTL+ formula over a finite name set $\mathcal{N}$ and a finite data set $\mathcal{D}$. We define $ABAR(\phi) = \langle Q, Rec_{\mathcal{N}}(\mathcal{D}), \rightarrow, Q_0, \mathcal{F}, V \rangle$ to be the generalized augmented Büchi automaton of records such that

- Q is the set of all atoms of ϕ,
- Q_0 is the set of atoms containing ϕ itself,
- the labeling function $V : Q \rightarrow (2^{\mathcal{N}} \rightarrow \{true, false\})$ is defined such that for all $q \in Q$ and $N \subseteq \mathcal{N}$, $V(q)(N) = true$ if and only if $N \in q$.
- the transition relation $\rightarrow \subseteq Q \times Rec_{\mathcal{N}}(\mathcal{D}) \times Q$ is defined such that $\forall p, q \in Q$ and for all $r \in Rec_{\mathcal{N}}(\mathcal{D})$ such that $dom(r) \subseteq N$ where N is the only set for which $V(q)(N) = true$, there is transition $q \xrightarrow{r} p$ if and only if
  - for all $\langle r' \rangle \psi \in q$, $r' = r$ and $\psi \in p$,
  - for all $\bigcirc \psi \in q$, $\psi \in p$,
  - for all $\neg \langle r' \rangle true \in q$, $r \neq r'$,

  (if for all $N \subseteq \mathcal{N}$, $V(q)(N) = false$ then only $r = \tau$ should be considered),
- $\mathcal{F}$ consists of the accepting sets

$$F_{\alpha \, U \, \beta} = \{q \in Q | \alpha \; U \; \beta \notin q \text{ or } \beta \in q\}$$

  for each $\alpha \, U \, \beta \in CL(\phi)$.

Before showing that the above construction is sound and complete, note that the resulting automaton is exactly an augmented BAR, namely the labeling function is so defined that for every transition $q \xrightarrow{r} p$ the label of q implies the weakest precondition of r. Also, note that each atom and thus each state q of the constructed automaton contains at most one of the sets of the form N. Thus, in each state q of the automaton there is at most one set N whose label is $true$, namely $V(q)(N) = true$.

The following theorem shows the correctness of the above construction:

Theorem 6.2 Let ϕ be a ρLTL+ formula over a names set $\mathcal{N}$ and a data set $\mathcal{D}$. The language accepted by $ABAR(\phi)$ is the set of all models of ϕ:

$$L(ABAR(\phi)) = \| \; \phi \; \| \; .$$

Proof.

Soundness ($L(ABAR(\phi)) \subseteq \| \; \phi \; \|$). Let $M = N_0 r_0 N_1 r_1 \cdots \in L(ABAR(\phi))$ be a guarded string accepted by the accepting computation $\pi = q_0 r_0 q_1 r_1 \cdots$ in automaton $ABAR(\phi)$. We show that for all $i \geq 0$ and every ρLTL formula ψ, if $\psi \in q_i$ then $M^i = N_i r_i N_{i+1} r_{i+1} \cdots \models \psi$. Using this fact and because $\phi \in q_0$ we obtain that $M \models \phi$ and thus $M \in \| \; \phi \; \|$.

The fact that for all $i \geq 0$ and every ρLTL+ formula ψ, if $\psi \in q_i$ then $M^i \models \psi$ is shown by induction on the structure of the formula ψ.

Base cases:

- $\psi = N$. Because $N \in q_i$, $V(q_i)(N) = true$. Using the facts that there is at most one set N for which $V(q_i)(N) = true$ and M is accepted by $ABAR(\phi)$, we know that $N_i = N$. Thus, $M^i \models \psi$.

- $\psi = \neg\langle r\rangle true$. Because $\neg\langle r\rangle true \in q_i$ we have $r_i \neq r$. Therefore, $M^i \models \psi$.

Inductive steps:

- $\psi = \psi_1 \vee \psi_2$. Because $\psi_1 \vee \psi_2 \in q_i$ using the definition of atoms we know that $\psi_1 \in q_i$ or $\psi_2 \in q_i$. By the induction hypothesis, $M^i \models \psi_1$ or $M^i \models \psi_2$. Thus, $M^i \models \psi$.

- $\psi = \psi_1 \wedge \psi_2$. The proof of this case is very similar to the previous case.

- $\psi = \bigcirc\psi_1$. Because $\bigcirc\psi_1 \in q_i$ using the definition of the transition relation we know that $\psi_1 \in q_{i+1}$. By the induction hypothesis, $M^{i+1} \models \psi_1$. Thus, $M^i \models \psi$.

- $\psi = \langle r\rangle\psi_1$. Because $\langle r\rangle\psi_1 \in q_i$ using the definition of the transition relation we know that $r_i = r$, $dom(r_i) \subseteq N_i$ and $\psi_1 \in q_{i+1}$. By the induction hypothesis, $M^{i+1} \models \psi_1$. Thus, $M^i \models \langle r\rangle\psi_1$.

- $\psi = [r]\psi_1$. Because $[r]\psi_1 \in q_i$ using the definition of atoms we know that $\langle r\rangle\psi_1 \in q_i$ or $\neg\langle r\rangle true \in q_i$:

  - If $\langle r\rangle\psi_1 \in q_i$ then by the proof of the previous case we know that $M^i \models \langle r\rangle\psi_1$. Thus, $M^i \models [r]\psi_1$.
  - If $\neg\langle r\rangle true \in q_i$ then using the base case, $M^i \models \neg\langle r\rangle true$. Thus, $M^i \models [r]\psi_1$.

- $\psi = \psi_1 \, U \, \psi_2$. Because $q_i\, q_{i+1} \cdots$ is an accepting run in the automaton, there is $k \geq i$ such that $q_k \in F_{\psi_1 \, U \, \psi_2}$. Let j be the least such k:

  - If $j = i$, then since $\psi_1 \, U \, \psi_2 \in q_i$ and $q_i \in F_{\psi_1 \, U \, \psi_2}$ using the definition of the final states we must have $\psi_2 \in p_i$. By the induction hypothesis, $M^i \models \psi_2$. Thus, $M^i \models \psi_1 \, U \, \psi_2$.
  - If $j > i$ then for all $i \leq l < j$, $\psi_1 \, U \, \psi_2 \in q_l$ and $\psi_2 \notin q_l$. Since q_i is an atom, $\psi_1 \in q_l$. By the induction hypothesis, for all $i \leq l < j$, $M^l \models \psi_1$. Now, $\psi_1 \, U \, \psi_2 \in q_{j-1}$ and $\psi_2 \notin q_{j-1}$, thus by the definition of atoms $\bigcirc(\psi_1 \, U \, \psi_2) \in q_{j-1}$. Therefore, $\psi_1 \, U \, \psi_2 \in q_j$. Since $q_j \in F_{\psi_1 \, U \, \psi_2}$ we should have $\psi_2 \in q_j$. By the induction hypothesis, $M^j \models \psi_2$. Thus we have for all $i \leq l < j$, $M^l \models \psi_1$ and $M^j \models \psi_2$. Therefore, $M^i \models \psi_1 \, U \, \psi_2$.

- $\psi = \psi_1 \, R \, \psi_2$. We have $\psi_1 \, R \, \psi_2 \in q_i$. By the definition of atoms, one of the following cases happens:

  - For all $j \geq i$, $\psi_2 \in q_j$ and $\psi_1 \, R \, \psi_2 \in q_j$. In this case by the induction hypothesis, for all $j \geq i$, $M^j \models \psi_2$. Thus, $M^i \models \psi$.
  - There is $j \geq i$ such that for all $i \leq l < j$, $\psi_2 \in q_l$, $\psi_1 \, R \, \psi_2 \in q_l$ and $\psi_1, \psi_2 \in q_j$. Then, for all $i \leq l < j$, $M^l \models \psi_2$ and $M^j \models \psi_1$ and $M^j \models \psi_2$. Thus, $M^i \models \psi$.

Completeness ($\| \phi \|_{\subseteq} \subseteq L(ABAR(\phi))$). Let the guarded string $M = N_0 r_0 N_1 r_1 \cdots$ be a model of ϕ. We show that $M \in L(ABAR(\phi))$. For this purpose for every $i \geq 0$ we define the set of formulas q_i as follows:

$$q_i = \{\psi \in CL(\phi) \mid M^i \models \psi\}.$$

Now we show that q_i's are atoms for ϕ and $\pi = q_0 r_0 q_1 r_1 \cdots$ is an accepting initial computation for M in $ABAR(\phi)$.

First note that each q_i satisfies the conditions to be an atom for ϕ (see Definition 6.6):

- (1) Obviously for all i, $true \in q_i$.
- (2) Let $N \in q_i$. Since $M^i \models N$, $N_i = N$. Thus, for all $N' \neq N$, $N' \neq N_i$. Therefore, for all $N' \neq N$, $M^i \not\models N'$. So, for all $N' \neq N$, $N \notin q_i$.
- (3) Let $\psi_1 \vee \psi_2 \in q_i$. Thus, $M^i \models \psi_1 \vee \psi_2$. Using the semantics of formulas, we have $M^i \models \psi_1$ or $M^i \models \psi_2$. Also, $\psi_1, \psi_2 \in CL(\phi)$. Thus, $\psi_1, \psi_2 \in q_i$.

The other conditions can be checked similarly.

Now, we show that for all $i \geq 0$, $q_i \xrightarrow{r_i} q_{i+1}$ is a transition in the automaton. For this purpose, we show that it satisfies the conditions of the transition relation in Definition 6.7. First note that since $M \models \phi$, we have the fact that $\forall i \geq 0, dom(r_i) \subseteq N_i$. Now we examine the conditions:

- Let $\langle r' \rangle \psi \in q_i$. Then, $M^i \models \langle r' \rangle \psi$. Thus, $r' = r_i$ and $M^{i+1} \models \psi$. Therefore, $r' = r_i$, $\psi \in q_{i+1}$, and $N_i \in q_i$ with $N_i \supseteq dom(r_i)$.
- Let $\bigcirc \psi \in q_i$. Then, $M^i \models \bigcirc \psi$. Thus, $M^{i+1} \models \psi$. Therefore, $\psi \in q_{i+1}$ and $N_i \in q_i$.
- Let $\neg \langle r' \rangle true \in q_i$. Then, $M^i \models \neg \langle r' \rangle true$. So $r_i \neq r'$ or $M^{i+1} \not\models true$. The second choice is impossible. Thus, $r_i \neq r'$.

So far, we have shown that π is a computation in the automaton $ABAR(\phi)$. Also, we know that π is an initial computation, because we have $M \models \phi$, thus $\phi \in q_0$. Therefore, $q_0 \in Q_0$.

Now, we show that π is a computation for the guarded string M. This fact is true because for each $i \geq 0$ the only $N \subseteq \mathcal{N}$ such that $M^i \models N$ and $N \in q_i$ is N_i. Thus, $\forall i \geq 0$, $V(q_i)(N_i) = true$. Thus, π is an initial computation for M.

Our proof is complete if we show that π is an accepting computation, namely that it meets at least one of the final states of every set of final states infinitely often. Suppose that it is not the case. Then, there is $j \geq 0$ such that for a formula of the form $\alpha U \beta$, we have $\forall k \geq j, q_k \notin F_{\alpha U \beta}$. Thus, $\forall k \geq j, \alpha U \beta \in q_k$ and $\beta \notin q_k$. So, $\forall k \geq j$, $M^k \models \alpha U \beta$ and $M^k \not\models \beta$. This contradicts the fact that $M^j \models \alpha U \beta$ since β never gets satisfied.

Therefore, π is an accepting initial computation for M in the automaton $ABAR(\phi)$. Thus, $M \in L(ABAR(\phi))$. □

The result reported in Theorem 6.2 can be used for an automata based procedure for model checking Reo connectors. Given an ABAR model B of a Reo connector, and a ρLTL formula ϕ over the same set of port names $\mathcal{N}$ and data set $\mathcal{D}$, saying that $B \models \phi$ is equivalent to check whether $L(B)$ does not contain any models of $\neg\phi$. From the above theorem, this is equivalent

to check if $L(B) \cap L(ABAR(\neg\phi)) = \emptyset$. Therefore, if this intersection is empty, it proves that the connector B satisfies the property ϕ. Otherwise, every element of this intersection is a counterexample. Recall that intersecting two Büchi automata is just a simple extension of the product construction, and checking for emptiness is decidable [138]. The complexity of the model checking procedure is linear in the number of states of B and exponential in the length of the formula ϕ [145].

6.3 On-the-fly translation

In this section, we sketch an algorithm to construct the ABAR for a ρLTL on-the-fly by generating the state space of the automaton incrementally, as required by the model checking procedure. The algorithm is a generalization of the on-the-fly approach proposed in [59] for standard LTL and extended with modalities for actions in a similar way as in [107].

6.3.1 A description of the algorithm

The algorithm works by building a graph underlying the ABAR to be defined for a formula ϕ. The nodes are labeled by sets of formulas that are obtained by decomposing them into their sub-formulas according to their boolean structures. Temporal formulas are handled by just deciding what should be true at the node and what must be true at every next node. For an on-the-fly construction of the graph, we need to store some information at every node of the graph. More specifically, a node is a structure containing the following fields:

1. *Name*. A string which is the name of the node.

2. *Incoming*. A set of elements of the form (q, X) where q is a node and $X \subseteq Rec_{\mathcal{N}}(\mathcal{D})$. Intuitively, a pair $(q, X) \in Incoming$ represents a transition from q to the current node labeled by the record r, for $r \in X$. A special element $init$ is used to mark initial nodes.

3. *Old*. A set of formulas that have already been processed and hold in the current node (provided the properties in *New* are satisfied).

4. *New*. A set of formulas that have not yet been processed and that have to be satisfied in the current node

5. $Next^{+}$. A set of next-state formulas that this node satisfies. They assert formulas that must be satisfied in any successor node.

6. $Next^{-}$. A set of records that are *not* allowed to label outgoing transitions from the current node.

The algorithm for building the graph of the automaton satisfying a ρLTL+ formula ϕ stores the nodes of the graph already computed in the list *Nodes_Set*. For all nodes in this list, it holds that the *New* field is empty. In this case, *Old* contains the set of formulas that the node

satisfies. The full graph can then be constructed using the information in the *Incoming* field of each node.

The algorithm starts with a node q_0 with its *New* field set to $\{\phi\}$, $Incoming = \{init\}$ and with all other fields initially set to empty. When processing a node q the algorithm removes a formula ψ from its *New* field and tries all possible ways to satisfy it, by looking at the syntactic structure of ψ:

- If $\psi = N$, where $N \subseteq \mathcal{N}$ then if there is N' ($N' \neq N$) in Old the node q is discarded because it contains a contradiction. Otherwise ψ is added to Old.

- If $\psi = \psi_1 \wedge \psi_2$ then both ψ_1 and ψ_2 are added to New because they both need to be satisfied in the node q.

- If $\psi = \psi_1 \vee \psi_2$ then a new node is created with the same fields as the current node q. Then ψ_1 is added to the *New* field of one node and ψ_2 to the other. The two nodes correspond to the two ways ψ can be satisfied.

- If $\psi = \bigcirc\varphi$ or $\psi = \langle r\rangle\varphi$ then ψ is added to the $Next^+$ field of the current node.

- The case where $\psi = [r]\varphi$ is novel with respect to the algorithm in [59]. Because $\psi \equiv \neg\langle r\rangle true \vee \langle r\rangle\varphi$, a new node is created with the same fields as the current node. The record r is added to the field $Next^-$ of one node, whereas the formula $\langle r\rangle\phi$ is added to the $Next^+$ field of the other node.

- If $\psi = \psi_1 \, U \, \psi_2$ then a new node is created with the same fields as the current node q. Because $\psi \equiv \psi_2 \vee (\psi_1 \wedge \bigcirc\psi)$, the formula ψ_2 is added to the *New* field of one node, while ψ_1 and $\bigcirc\psi$ are added to the fields *New* and $Next^+$ of the other node, respectively.

- If $\psi = \psi_1 \, R \, \psi_2$ then a new node is created with the same fields as the current node q. Because $\psi \equiv \psi_2 \wedge (\psi_1 \vee \bigcirc\psi)$, the formula ψ_2 is added to the *New* field of both nodes, ψ_1 is added to the *New* field of one node and $\bigcirc\psi$ to the $Next^+$ of the other node.

When the New field is empty, the current node is ready to be added to the set *Nodes_Set*. If there is already another node in the list with the same *Old*, $Next^+$, and $Next^-$ fields, then the only *Incoming* field of the copy that already exists needs to be updated by adding the edges in the *Incoming* field of the current node.

If there is no such node, then the current node is added to the list *Nodes_Set*, but different than the case of the original algorithm [59], there are several ways how a current node is formed for its successors: if the information about the labels of the outgoing transitions is inconsistent (i.e. $Next^+$ is empty or there is a record r in $Next^-$ that is also used in a next state formula $\langle r\rangle\varphi$ in $Next^+$ or there are two formulas $\langle r\rangle\varphi$ and $\langle r'\rangle\varphi'$ in $Next^+$ with $r \neq r'$) then there is no successor node.

Otherwise, if the formulas in the $Next^+$ field of the current node are only of type $\bigcirc\varphi$, then a successor node is created with a transition from the current node to the new node labeled by r for each record r not in the $Next^-$ field of the current node. The formulas to be satisfied by this new node are all formulas in the $Next^+$ field of the current node stripped off of their next state modality.

η	New_1	New_2	$Next_1$
$\psi_1 \vee \psi_2$	$\{\psi_1\}$	$\{\psi_2\}$	$\emptyset$
$\psi_1 \, U \, \psi_2$	$\{\psi_1\}$	$\{\psi_2\}$	$\{\bigcirc(\psi_1 \, U \, \psi_2)\}$
$\psi_1 \, R \, \psi_2$	$\{\psi_2\}$	$\{\psi_1, \psi_2\}$	$\{\bigcirc(\psi_1 \, R \, \psi_2)\}$

Table 6.1: Definitions of New_1, New_2 and $Next_1$ functions.

Finally, in the remaining case that there is a formula $\langle r \rangle \phi$ in $Next^+$ with no r in the $Next^-$ field, then a successor node is created with a transition labeled by r from the current node to the new node. As in the previous case, the formulas to be satisfied by this new node are all formulas in the $Next^+$ field of the current node stripped off of their next state modality.

6.3.2 The algorithm in detail

In this section we present the pseudo code of the algorithm sketched in the previous subsection. The algorithm constructs a graph of nodes and is called *Create_Graph*. It uses the function *Expand* which processes every node and updates the list of nodes *Nodes_Set*. For conciseness, we use functions New_1, New_2 and $Next_1$ which are defined in Table 6.1.

```
Creat_Graph(φ)
1.   return(Expand([Name: = New_Name(), Incoming: = {Init},
2.          New: = {φ}, Old: = ∅, Next+: = ∅, Next−: = ∅], ∅));
```

```
Expand(Node, Nodes_Set)
1. if New(Node) = ∅
2.    then if ∃ND ∈ Nodes_Set with Old(ND) = Old(Node) and
3.         Next+(ND) = Next+(Node) and Next−(ND) = Next−(Node)
4.         then { Incoming(ND): = Incoming(ND) ∪ Incoming(Node);
5.                return(Nodes_Set); }
6.         else if (∃ ⟨r⟩φ, ⟨r′⟩ψ ∈ Next+(Node) with r ≠ r′) or
7.                 (∃r ∈ Next−(Node), ⟨r⟩φ ∈ Next+(Node))
8.                 then return(Nodes_Set ∪ {Node})
9.                 else if ∄⟨r⟩φ ∈ Next+(Node)
10.                        then return(Expand([Name: = New_Name(),
11.                             Incoming: = {(Name(Node),
12.                                  Rec_N(D)\Next−(Node))}
13.                             New: = StriptNexts(Next+(Node))
14.                             old: = ∅
15.                             Next+: = ∅, Next−: = ∅],
16.                             Nodes_Set ∪ {Node}))
17.                        else return(Expand([Name: = New_Name(),
```

$Incoming := \{(Name(Node), \{r\})\},$
$New := StriptNexts(Next^+(Node)), Old := \emptyset$
$Next^+ := \emptyset, Next^- := \emptyset], Nodes\_Set \cup \{Node\}))$

else let $\eta \in New(Node)$
  then $New(Node) := New(Node) \backslash \{\eta\}$;
  switch
    case $\eta = N$ or $\eta = true$ or $\eta = false$:
      if $\eta = false$ or $\exists N' \in Old(Node)$ that $N' \neq N$
        then return$(Nodes\_Set)$
        else $\{Old(Node) := Old(Node) \cup \{\eta\}$;
          return$(Expand(Node, Nodes\_Set))\}$

    case $\eta = \phi\, U\, \psi$ or $\eta = \phi\, R\, \psi$ or $\eta = \phi \vee \psi$:
      $Node_1 := [Name := New\_Name(), Incoming := Incoming(Node),$
        $New := New(Node) \cup (New_1(\eta) \backslash Old(Node)),$
        $Old := Old(Node) \cup \{\eta\},$
        $Next^+ := Next^+(Node) \cup Next_1(\eta), Next^- := Next^-(Node)]$
      $Node_2 := [Name := New\_Name(), Incoming := Incoming(Node),$
        $New := New(Node) \cup (New_2(\eta) \backslash Old(Node)),$
        $Old := Old(Node) \cup \{\eta\},$
        $Next^+ := Next^+(Node), Next^- := Next^-(Node)]$
      return$(Expand(Node_2, Expand(Node_1, Nodes\_Set)))$

    case $\eta = \phi \wedge \psi$:
        $Old(Node) := Old(Node) \cup \{\eta\},$
        $New(Node) := New(Node) \cup (\{\phi, \psi\} \backslash Old(Node))$
      return$(Expand(Node, Nodes\_Set))$

    case $\eta = X\phi$ or $\eta = \langle r \rangle \phi$:
        $Old(Node) := Old(Node) \cup \{\eta\},$
        $Next^+(Node) := Next^+(Node) \cup (\eta\}$
      return$(Expand(Node, Nodes\_Set))$

    case $\eta = [r]\phi$:
      $Node_1 := [Name := New\_Name(), Incoming := Incoming(Node),$
        $New := New(Node), Old := Old(Node) \cup \{\eta\},$
        $Next^+ := Next^+(Node),$
        $Next^- := Next^-(Node) \cup \{r\}]$
      $Node_2 := [Name := New\_Name(), Incoming := Incoming(Node),$
        $New := New(Node), Old := Old(Node) \cup \{\eta\},$
        $Next^+ := Next^+(Node) \cup \{\langle r \rangle \phi\},$
        $Next^- := Next^-(Node)]$
      return$(Expand(Node_2, Expand(Node_1, Nodes\_Set)))$.

In the above algorithm, for each set of ρLTL+ formulas S we define:

$$StripNexts(S) = \{\phi \mid \bigcirc \phi \in S \text{ or } \langle r \rangle \phi \in S \text{ for some } r \in Rec_{\mathcal{N}}(\mathcal{D})\}.$$

6.3.3 The ABAR defined by the algorithm

The above sketched algorithm defines for each ρLTL+ formula ϕ a generalized ABAR $B(\phi)$ over port names $\mathcal{N}$ and data set $\mathcal{D}$ as follows. The states are the set of nodes in *Nodes_Set*, as returned by the algorithm. Every node with the *Init* in its *Incoming* field is an initial state. In each node (state) n, if there is (only one) $N \in Old(n)$ then the valuation function $V_B(n)$ assigns $true$ only to N, otherwise for all $N \subseteq \mathcal{N}$, $V_B(n)(N)$ is $true$. Note that for each node n at most one set $N \subseteq \mathcal{N}$ is in $Old(n)$ and for this N we have $V(n)(N) = true$. The transitions of the form $n \xrightarrow{r} n'$ are exactly those where $r \in X$ for (n, X) in the *Incoming* field of n' and $dom(r) \subseteq N$ and N is the only N for which $V(n)(N) = true$. Finally, for each sub-formula $\alpha U \beta$ of ϕ we define a set of accepting states $F_{\alpha U \beta}$ containing all nodes n for which $\alpha \ U \ \beta \notin t(q)$ or $\beta \in t(n)$, where $t(n)$ is the union of the fields *Old*, *New*, $Next^+$ and the set containing $\neg\langle r \rangle true$ for each record r in the $Next^-$ field of the node n:

$$t(n) = New(n) \cup Old(n) \cup Next^+(n) \cup \{\neg\langle r \rangle true \mid r \in Next^-(n)\}.$$

(Note that here we require function t to be defined only on finished nodes that belong to $Node\_Set$ and whose New field is empty. Our definition is more general because we want to use it in the next proofs.)

More formally, we define ABAR $B(\phi)$ as follows:

Definition 6.8 Let ϕ be a ρLTL+ formula over a finite name set $\mathcal{N}$ and a finite data set $\mathcal{D}$. We define $B(\phi) = \langle Q_B, Rec_{\mathcal{N}}(\mathcal{D}), \rightarrow_B, Q_{0B}, \mathcal{F}_B, V_B \rangle$ to be the generalized augmented Büchi automaton of records such that

- $Q_B = Node\_Set$ is the set of all nodes generated by the algorithm,
- $Q_{0B} = \{n \mid n \in Node\_Set \text{ and } Init \in Incoming(n)\}$,
- the labeling function $V_B : Node\_Set \rightarrow (2^{\mathcal{N}} \rightarrow \{true, false\})$ is defined such that for all $n \in Node\_Set$, if there exists an $N \subseteq \mathcal{N}$ such that $N \in Old(n)$ then $V_B(n)(N) = true$ otherwise $\forall N \subseteq \mathcal{N}$, $V_B(n)(N) = true$,
- the transition relation $\rightarrow_B \subseteq Node\_Set \times Rec_{\mathcal{N}}(\mathcal{D}) \times Node\_Set$ is defined such that $\forall n, n' \in Node\_Set$ and $\forall r \in Rec_{\mathcal{N}}(\mathcal{D})$, we have $n \xrightarrow{r}_B n'$ if and only if $\exists (n, X) \in Incoming(n')$ such that $r \in X$ and $V_B(n)(dom(r)) = true$,
- $\mathcal{F}'_B$ consists of the accepting sets

  $$F'_{\alpha U \beta} = \{n \in Node\_Set \mid \alpha \ U \ \beta \notin t(n) \text{ or } \beta \in t(n)\}$$

  for each $\alpha U \beta \in CL(\phi)$.

Theorem 6.3 Let ϕ be a ρLTL+ formula over a finite names set $\mathcal{N}$ and a finite data set $\mathcal{D}$ and $B(\phi)$ be the ABAR produced by the above algorithm. Then, the accepted language of $B(\phi)$ is the set of all models of ϕ, that is

$$L(B(\phi)) = \| \phi \| \ .$$

Proof. Obviously this theorem is correct if we show that both automata $ABAR(\phi)$ (which we construct it globally) and $B(\phi)$ accept precisely the same language, namely $L(ABAR(\phi)) = L(B(\phi))$. We will prove this fact in Section 6.3.4 after presenting some lemmas. □

As explained before, a formula about a Reo connector can be verified by (1) constructing the ABAR translation of negation of the formula, (2) constructing the product automaton using the ABAR model of the Reo connector, and (3) checking the resulting automaton for emptiness.

6.3.4 Proof of the correctness

In this section we prove Theorem 6.3 in detail. As we mentioned in the theorem's proof scheme, we will show that for a ρLTL+ formula ϕ both automata $ABAR(\phi)$ (which we construct globally as in Definition 6.7) and $B(\phi)$ (which we construct by the on-the-fly algorithm as in Definition 6.8) accept precisely the same language, namely $L(ABAR(\phi)) = L(B(\phi))$.

Soundness ($L(B(\phi)) \subseteq L(ABAR(\phi))$).

First we present a simple lemma:

Lemma 6.4 Let $n \in Node\_Set$ be a node generated by the algorithm and $t(n)$ be the set of formulas for n as we defined in section 6.3.3. Also, define the set of formulas A^n as:

$$A^n = t(n) \cup \{true\}.$$

Then for each node n, A^n is an atom of ϕ.

Proof. First, clearly A^n is a subset of $CL(\phi)$. Now, refer to conditions (1) to (7) in Definition 6.6 which must be satisfied for a set of formulas to be an atom. Simply, we can show that all of them are satisfied by A^n. Based on the definition of A^n we know that $true \in A^n$ and since $false$ is never part of any node (see lines 25-26 of the *Expand* algorithm in Section 6.3.2), $false \notin A^n$. Thus A^n satisfies condition (1). Lines 24-28 of the *Expand* algorithm show that for all $N \subseteq \mathcal{N}$, $N \in A^n$ if and only if for all $N' \neq N$, $N' \notin A^n$. Thus condition (2) is also satisfied by A^n. For conditions (3) to (7), note that whenever a formula on the left hand side of these conditions are inserted into Old, the required formulas get inserted into New and these formulas will eventually get into Old and hence into A^n. For example, for condition (3), if $\phi_1 \vee \phi_2 \in A^n$ then it should be in $t(n)$. Thus, when processed, either ϕ_1 or ϕ_2 gets inserted into New. But each formula inserted into New will eventually get into all finished nodes under this node. Thus, A^n will have either ϕ_1 or ϕ_2. Similarly, other conditions can be verified. □

Now, we can prove the soundness lemma:

Lemma 6.5 Let ϕ be a ρLTL+ formula. Then,

$$L(B(\phi)) \subseteq L(ABAR(\phi)).$$

Proof. Let $M = N_0 r_0 N_1 r_1 \cdots \in L(B(\phi))$ be accepted by an accepting computation $\sigma = n_0 r_0 n_1 r_1 \cdots$ in $B(\phi)$. Thus, we know that $\forall i$, $V_B(n_i)(N_i) = true$. Consider $A_i = t(n_i) \cup \{true\}$. First, by Lemma 6.4, for all i, A_i is an atom. Also, it is clear that in $ABAR(\phi)$, $\forall i$, $V(n_i)(N_i) = true$.
We will show that $\pi = A_0 r_0 A_1 r_1 \cdots$ is an accepting computation for M in $ABAR(\phi)$:
1- We know that $t(n_0) \subseteq A_0$ and $\phi \in t(n_0)$. Hence, $\phi \in A_0$. Thus, $A_0 \in Q_0$.
2- Now consider the transition $n_i \xrightarrow{r_i}_B n_{i+1}$ in $B(\phi)$. First note that $\forall i, dom(r_i) \subseteq N_i$. Since in the *Expand* algorithm n_{i+1} was spawned from n_i, it is clear that $\langle r \rangle \psi \notin A_i$ (where $r \neq r_i$) and $r_i \notin Next^-(n_i)$. Now, for all $\bigcirc\psi \in t(n_i)$, $\psi \in t(n_{i+1})$ (by construction) and for all $\langle r_i \rangle \psi \in t(n_i)$, $\psi \in t(n_{i+1})$. Hence we have:

- for all $\langle r \rangle \psi \in A_i$, $r = r_i$,
- for all $\bigcirc\psi \in A_i$, $\psi \in A_{i+1}$,
- for all $\langle r_i \rangle \psi \in A_i$, $\psi \in A_{i+1}$ and
- $\neg\langle r_i \rangle true \notin A_i$.

Therefore, $\forall i$, $A_i \xrightarrow{r_i} A_{i+1}$ is a transition in $ABAR(\phi)$.
3- Also, if $n_i \in F'_{\alpha U \beta}$, then either $\alpha U \beta \notin t(n_i)$ or $\beta \in t(n_i)$. If $\alpha U \beta \notin t(n_i)$, then $\alpha U \beta \notin A_i$. If $\beta \in t(n_i)$, then $\beta \in A_i$. Thus, $A_i \in F_{\alpha U \beta}$. Since σ meets each set $F'_{\alpha U \beta}$ infinitely often, π meets each set $F_{\alpha U \beta}$ infinitely often.

By the above 1-3 facts, we conclude that π is an initial accepting computation for M in $ABAR(\phi)$. Therefore, $M \in L(ABAR(\phi))$. □

Completeness ($L(ABAR(\phi)) \subseteq L(B(\phi))$).
Now, we show that for every ρLTL+ formula ϕ, each model accepted by $ABAR(\phi)$ is also accepted by $B(\phi)$. We do this by mapping accepting computations of $ABAR(\phi)$ to accepting computations over the algorithm automaton $B(\phi)$. First we present a definition.

Let $n \in Node\_Set$ be a node constructed by the *Expand* algorithm starting with formula ϕ. We define $f(n)$ to be the set of all atoms for ϕ that can extend node n. More formally:

$$f(n) = \{A \mid \text{A is an atom for } \phi \text{ and } t(n) \subseteq A\}.$$

Now, we give some lemmas that will lead us to the proof of completeness.

Lemma 6.6 When a node n is split in the algorithm into two nodes n_1 and n_2 (lines 30-39 and 51-58) the following holds:

$$f(n) = f(n_1) \cup f(n_2).$$

Similarly, when a node n is updated to become a new node n' (lines 24-28 and 41-49) the following holds:

$$f(n) = f(n').$$

Proof. The proof is obvious by tracing of the algorithm and calculating $t(n')$ for every new node n' using $t(n)$. □

When a node n is spawned with its Old, $Next^+$ and $Next^-$ fields empty, the algorithm starts processing the formulas in New. From this point onwards, one can view the algorithm as creating a tree rooted at n. The tree gets modified as formulas in New are processed. When a node (which must be a leaf) splits, we can view this as a creation of two children, since the algorithm will start expanding each child eventually. When a node is processed and its fields get modified, we view this as the creation of a single child. When a node is abandoned, we mark the node *bad*, no new edge comes out of this node. Finally a tree is produced which is rooted at n. We will call a leaf node *good* if it is not bad. Note that the good leaves at the end of the construction have their New fields empty and are exactly the nodes added to $Nodes\_Set$. The proof of the following lemma is by induction on the number of steps that have been performed so far by the algorithm, which have modified the tree.

Lemma 6.7 Let n be a node and A an atom such that $A \in f(n)$. At any point in the construction of the tree rooted at node n, there is a good leaf n' such that $A \in f(n')$ and for all formulas of the form $\alpha U \beta$ in $CL(\phi)$, if $A \in F_{\alpha U \beta}$ then $\alpha U \beta \notin Old(n')$ or $\beta \in t(n')$.

Proof. The proof can be simply done by induction on the number of steps in the algorithm that have changed the tree so far. □

Lemma 6.8 Let n be a rooted node and A an atom such that $A \in f(n)$. Then, there is a good leaf n' in the tree rooted by n such that $A \in f(n')$ and for all formulas of the form $\alpha U \beta$ in $CL(\phi)$, if $A \in F_{\alpha U \beta}$ then $n' \in F'_{\alpha U \beta}$.

Proof. By Lemma 6.7, at the end of the construction of the tree, there exists a leaf n' such that $A \in f(n')$ and for all formula of the form $\alpha U \beta$ in $CL(\phi)$, if $A \in F_{\alpha U \beta}$ then $\alpha U \beta \notin Old(n')$ or $\beta \in t(n')$. Since $New(n') = \emptyset$ and n' is a good leaf, thus for all formula of the form $\alpha U \beta$ in $CL(\phi)$, $\alpha U \beta \notin t(n')$ or $\beta \in t(n')$. Therefore, using Definition 6.8, $n' \in F'_{\alpha U \beta}$. □

Lemma 6.9 Let n be a node, A an atom that $A \in f(n)$, and let $A \xrightarrow{r} A'$ be a transition in $ABAR(\phi)$. Then, there is an $n' \in Node\_Set$ such that there is transition $n \xrightarrow{r} n'$ in $B(\phi)$ where, $A' \in f(n')$ and for all formulas of the form $\alpha U \beta$ in $CL(\phi)$, if $A' \in F_{\alpha U \beta}$ then $n' \in F'_{\alpha U \beta}$.

Proof. First, it is clear that there is no $\neg\langle r \rangle true$ or $\langle r' \rangle \psi$ $(r' \neq r)$ in $t(n)$, for otherwise it would belong to A as well and $A \xrightarrow{r} A'$ will not be possible in $ABAR(\phi)$. Hence, after n was processed by the algorithm, a node m must have been spawned with its New field set contains the formulas in $Next^+(n)$ stripped of their $\bigcirc$'s and $\langle r \rangle$'s. Now, if$\psi \in t(m)$ then $\bigcirc \psi \in t(n)$ or $\langle r \rangle \psi \in t(n)$. Then, $\bigcirc \psi \in A$ or $\langle r \rangle \psi \in A$. Thus, $\psi \in A'$. Hence, $t(m) \subseteq A'$ and $A' \in t(m)$. By Lemma 6.8, there exists a good leaf, and hence a node in $Node\_Set$, say n', in the tree rooted at m, such that $A' \in f(n')$ and for all formulas of the form $\alpha U \beta$ in $CL(\phi)$, if $A' \in F_{\alpha U \beta}$ then $n' \in F'_{\alpha U \beta}$. Also, there exists $(n, X) \in Incoming(n')$ such that $r \in X$. Hence $n \xrightarrow{r} n'$ in $B(\phi)$. This completes the proof. □

Lemma 6.10 Let A_0 be an initial atom (state) in $ABAR(\phi)$, namely $A_0 \in Q_0$. Then, there is a node n_0 in $B(\phi)$, namely $n_0 \in Q_{0B}$, such that $A_0 \in f(n_0)$.

Proof. The algorithm starts with a node m with $New(m) = \{\phi\}$. Since $A_0 \in Q_0$, $\psi \in A_0$. Thus, $t(m) \subseteq A_0$ and $A_0 \in f(m)$. Lemma 6.8 guarantees the existence of some good leaf n_0 which hence gets into $Node\_Set$, in the tree rooted at m such that $A_0 \in f(n_0)$. Also, any leaf of the tree rooted at m has $Init \in Incoming$ and hence $n_0 \in Q_{0B}$. □

Lemma 6.11 Let ϕ be a ρLTL+ formula. Then,

$$L(ABAR(\phi)) \subseteq L(B(\phi)).$$

Proof. Let $M = N_0 r_0 N_1 r_1 \cdots \in L(ABAR(\phi))$ and let $\pi = A_0 r_0 A_1 r_1 \cdots$ be an accepting initial computation for it in $ABAR(\phi)$. We exhibit an accepting initial computation of $B(\phi)$ that accepts M. First, by Lemma 6.10, there exists $n_0 \in Q_{0B}$ such that $A_0 \in f(n_0)$. We construct an accepting initial computation $n_0 r_0 n_1 r_1 \cdots$ for M by using Lemma 6.9 repeatedly, Assume that we have constructed a partial computation $n_0 r_0 \cdots n_i$ so far such that $A_i \in f(q_i)$ (this is true in the beginning for the partial computation n_0). Now since $A_i \xrightarrow{r_i} A_{i+1}$ and $A_i \in f(q_i)$ using Lemma 6.9, there exists n_{i+1} such that $n_i \xrightarrow{r_i} n_{i+1}$ and $A_{i+1} \in f(n_{i+1})$ and for all formulas of the form $\alpha U \beta$ in $CL(\phi)$, if $A_{i+1} \in F_{\alpha U \beta}$ then $n_{i+1} \in F'_{\alpha U \beta}$.

Thus we have extended the partial computation to $n_0 r_0 \cdots n_{i+1}$ with $A_{i+1} \in f(n_{i+1})$. Continuing in this fashion, we can build an infinite initial computation $\rho = n_0 r_0 n_1 r_1 \cdots$ such that $\forall j, A_j \in f(n_j)$ and for all formulas of the form $\alpha U \beta$ in $CL(\phi)$, if $A_j \in F_{\alpha U \beta}$ then $n_j \in F'_{\alpha U \beta}$. Since π meets every final set infinitely often, the computation ρ also meets all final sets infinitely often. Hence ρ is an accepting computation in $B(\phi)$.

The proof is complete if we show that $\forall j$, $V_B(n_j)(N_j) = true$. Since π is an accepting computation for M in $ABAR(\phi)$, we have $\forall j$, $V(A_j)(N_j) = true$. Thus, by the definition of the V function in $ABAR(\phi)$, we know that for all j, $N_j \in A_j$ and $\forall N \neq N_j, N \notin A_j$. Thus, if there exists $N \in Old(n_j)$ then $N \in t(n_j)$. Thus, $N = N_j$ and $V_B(n_j)(N_j) = true$. Otherwise, if there is no $N \in Old(n_j)$, by the definition of V_B, $\forall N$, $V_B(n_j)(N) = true$. Therefore, in both cases we have that $V_B(n_j)(N_j) = true$. Hence the lemma is proved. □

By Lemmas 6.5 and 6.11, we have a complete proof for Theorem 6.3. This shows that our on-the-fly construction of the automaton is correct.

7 A Reo Model Checker

In this chapter, we introduce the main theoretical and practical concepts we used to implement a binary decision diagrams (BDD) based model checking tool for Reo specifications. This implementation is based on the augmented Büchi automata of records semantic model introduced in the previous chapters. Moreover, this tool accepts properties expressed in the ρLTL linear temporal logic as input and verifies a Reo specification against these properties. The Reo language has a wide range of applicability in coordination modeling and many real world case studies need a large number of channels to model the sophisticated orchestration patterns among constituent components. To address complex coordination patterns in large Reo circuits our proposed solution is based on BDDs.

Applying BDDs and using a symbolic representation for the underlying state space, helps us to improve the performance in small and middle size cases and also expands the applicability of our tool to larger Reo circuits. In the remainder of this chapter we first introduce a method for encoding of an ABAR as BDDs and reformulating of ABAR join operation in BDD terms. Next, we propose a method for converting a ρLTL formula to its equivalent Büchi automata and also apply the previously described procedure to represent the automata with BDDs. Having the BDD representation of an underlying ABAR of a Reo circuit and the BDD representation of a ρLTL property, we explain our model checking procedure in the following section. Finally, we present some experimental results of our tool.

7.1 Binary Decision Diagrams

Binary decision diagrams are data structures used for compressed representation of switching functions or Boolean formulas [111, 50, 29]. They are often more compact than other ways of representing of Boolean formulas, such as conjunctive and disjunctive normal forms, and they can be manipulated more efficiently [50]. In this section, we briefly describe binary decision diagrams and review their preliminaries. Our presentation and notations in this section is based on the textbooks [29, 50].

Definition 7.1 Let $Var = \{x_1, \ldots, x_n\}$ be a set of Boolean variables and $Eval(Var)$ be the set of all evaluations for $x_1, \ldots, x_n$, that is, the set of all total functions of the type $Var \rightarrow \{0, 1\}$. A *switching function* for $Var = \{x_1, \ldots, x_n\}$ is a function $f : Eval(Var) \rightarrow \{0, 1\}$. The switching functions for the empty variable set ($Var = \emptyset$) are just constants 0 or 1.

Obviously, each Boolean formula over the Boolean propositions $x_1, \ldots, x_n$ represents a switching function and vise versa.

Definition 7.2 Let $f : Eval(z, y_1, \ldots, y_m) \longrightarrow \{0, 1\}$ be a switching function. The *positive cofactor* of f with respect to variable z is the switching function $f|_{z=1}$ of the type $Eval(y_1, \ldots, y_m) \longrightarrow \{0, 1\}$ in which for all m-tuple of bits $(b_1, \ldots, b_m)$,

$$f|_{z=1}(b_1, \ldots, b_m) = f(1, b_1, \ldots, b_m).$$

Similarly, the *negative cofactor* of f with respect to variable z is the switching function $f|_{z=0}$ of the type $Eval(y_1, \ldots, y_m) \longrightarrow \{0, 1\}$ in which for all m-tuple of bits $(b_1, \ldots, b_m)$,

$$f|_{z=0}(b_1, \ldots, b_m) = f(0, b_1, \ldots, b_m).$$

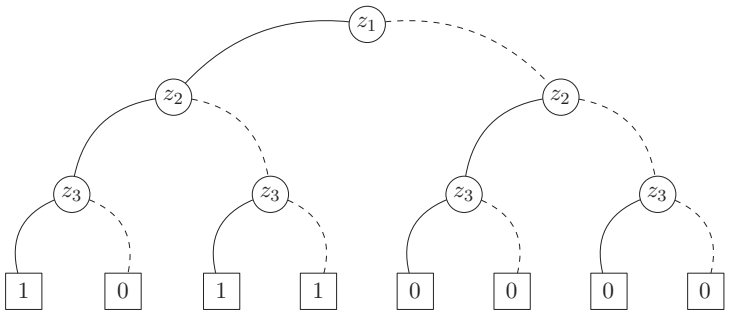


Figure 7.1: Binary decision tree for switching function $f = z_1 \wedge (\neg z_2 \vee z_3)$ [29].

The following result shows how a switching function f can be decomposed into its cofactors, called *Shannon expansion* [29]:

Lemma 7.1 If f is a switching function for the set of variables Var, then for each variable $z \in Var$,

$$f = (\neg z \wedge f|_{z=0}) \vee (z \wedge f|_{z=1}).$$

Using the Shannon expansion, one can represent switching functions by *binary decision trees*: Let f be a switching function for some variable set Var and fix an arbitrary enumeration $x_1, \ldots, x_n$ for the variables in Var. Now, we can represent f using a binary tree of height n such that the two outgoing edges of the inner nodes at level i stand for the cases $x_i = 0$ (depicted by a dashed line) and $x_i = 1$ (depicted by a solid line). Thus, the paths from the root to a leaf in that tree represent the evaluations and their corresponding values. The leaves (terminal nodes represented by boxes) stand for the function values 0 or 1 of f.

As an example, the binary decision tree for switching function $f(z_1, z_2, z_3) = z_1 \wedge (\neg z_2 \vee z_3)$ appears in Figure 7.1.

Binary decision trees are quite close to the representation of Boolean functions as truth tables as far as their sizes are concerned. However, they often contain some redundancy which we can exploit. Since 0 and 1 are the only terminal nodes of binary decision trees, we can optimize the representation by having pointers to just one copy of 0 and one copy of 1. Also, for more optimization we can remove unnecessary decision points in the tree, collapse constant subtrees (i.e., subtrees all whose terminal nodes have the same value) into a single node and merge isomorphic subtrees. This results in a directed acyclic graph (DAG) called a *binary decision diagram* (BDD).

As an example, the binary decision diagram obtained from the binary decision tree illustrated in Figure 7.1 is given in Figure 7.2.

A representation of switching functions is called a *canonical* representation if it satisfies the property that two switching functions are logically equivalent if and only if they have isomorphic representations [50]. This property simplifies tasks like checking equivalence of two formulas and deciding if a given formula is satisfiable or not. Bryant [40] showed how to obtain a canonical representation for switching functions by placing two restrictions on binary

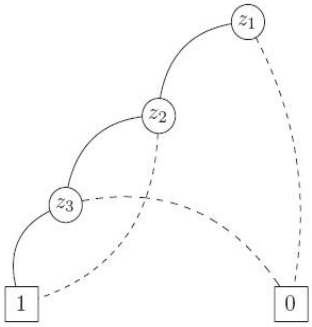


Figure 7.2: Binary decision diagram for switching function $f = z_1 \wedge (\neg z_2 \vee z_3)$ [29].

decision diagrams. First, the variables should appear in the same order along each path from the root to a terminal. Second, there should be no isomorphic subtrees or redundant vertices in the diagram. The first requirement is achieved by imposing a total ordering on the variables and the second is achieved by repeatedly applying three transformation rules that do not alter the function represented by the diagram [40, 50]:

- *Remove duplicate terminals*. Eliminate all but one terminal vertex with a given label and redirect all arcs to the eliminated vertices to the remaining one.

- *Remove duplicate nonterminals*. If two nonterminal nodes u and v have the same label and their both subtrees are equal with each other, then eliminate u or v and redirect all incoming arcs to the other vertex.

- *Remove redundant tests*. If the left and right subtrees of a nonterminal node v are the same, then eliminate v and redirect all of its incoming arcs to the root of the subtree of v.

Starting with a binary decision diagram satisfying the ordering property, its canonical form is obtained by applying the above transformation rules until the size of the diagram can no longer be reduced. Bryant shows how this can be done in a bottom-up manner which is linear in the size of the original binary decision diagram [40]. The term *ordered binary decision diagram* (OBDD) will be used to refer to the graph obtained in this manner [50].

The size of an OBDD can depend critically on the selected variable ordering. In general, finding an optimal ordering for a set of variables is infeasible; in fact, it can be shown that even checking that a particular ordering is optimal is NP-complete. Moreover, there are Boolean functions that have exponential size OBDDs for any variable ordering. Several heuristics have been developed for finding a good variable ordering when such an ordering exists [50].

7.2 Encoding ABARs as BDDs

In this section we introduce a symbolic representation method for the ABAR model introduced in previous chapters. As our final aim is to implement a BDD based model checker, the state space generation should be transformed to BDD terms. Therefore, in our next step we propose an operation to mimic the join operation of ABAR models in BDD domain.

In the first step of our model checking procedure we represent the ABAR corresponding to each Reo channel in a symbolic way. Let $B = \langle Q, \Sigma, \longrightarrow, Q_0, F, l\rangle$ be an ABAR, where Σ is a set of records over a finite set of port names $\mathcal{N}$ and a finite data set $\mathcal{D}$. The ABAR B can be represented by a tuple $E_B = \langle \mathcal{N}_B, Q_B, \longrightarrow_B, Q_{B_0}, F_B\rangle$ of Boolean expressions which encodes port names, states and their labels, transition relation, initial states, and final states, respectively. In the following paragraphs, we present the encodings step by step.

- Let $V = \{n_1, \ldots, n_j\}$ be considered as a set of Boolean propositions corresponding to each port in $\mathcal{N}$. We represent $\mathcal{N}$ by the following Boolean expression:

$$\mathcal{N}_B = \bigvee_{n \in V} n.$$

- To symbolically represent the set of states Q, we use a set of Boolean state propositional variables $V_q = \{q_1, \ldots, q_k\}$, where $k = \lceil \log_2(|Q|)\rceil$, and assign a unique evaluation of $(q_1, \ldots, q_k) = (b_1, \ldots, b_k)$ to each state where $b_i \in \{0, 1\}$. Each state $q \in Q$ can be uniquely identified by the Boolean expression,

$$\phi_q = l(q) \wedge (\bigwedge_{1 \leq i \leq k} q_i)$$

  where, $l(q)$ is the propositional label of the state q. Also, the set of states Q is represented by the following Boolean expression:

$$Q_B = \bigvee_{q \in Q} \phi_q.$$

  Where we know that for all $q, q' \in Q$ it is the case that $l(q) \not\models l(q')$, we simply represent each state by its label ($\phi_q = l(q)$) and the set of all states Q by

$$Q_B = \bigvee_{q \in Q} l(q).$$

  Initial and final states are encoded in a similar way.

- Let $r = [n_1 = d_1, \ldots, n_i = d_i]$ be a record. It can be represented as the Boolean expression,

$$r_B = (\bigwedge_{n \in dom(r)} com_n) \bigwedge (\bigwedge_{n \in \mathcal{N} \setminus dom(r)} \neg com_n) \bigwedge (\bigwedge_{(n,d) \in r} \psi_{n,d})$$

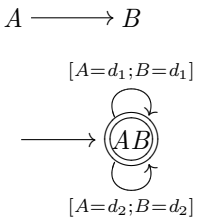


Figure 7.3: A synchronous channel and its ABAR model

where, for each $n \in \mathcal{N}$, com_n is a Boolean variable that intuitively says that port n is communicating and for each $(n, d) \in (\mathcal{N} \times \mathcal{D})$, $\psi_{n,d}$ is a Boolean variable that intuitively says that the data item in port n is d.

- Symbolic representation of a single transition involves the encoding of its source state, symbolic representation of its corresponding record label, and the label of its target state. To distinguish the source and the target states of a single transition, we use the above mentioned set of variables $V_q = \{q_1, \ldots, q_k\}$ and the set of ports $N = \{n_1, \ldots, n_j\}$ for the source state and a primed version of them, $V'_q = \{q'_1, \ldots, q'_k\}$ and $N' = \{n'_1, \ldots, n'_j\}$, for the target state. Therefore, the transition $T = q \xrightarrow{r} p$ is encoded by

$$T_B = \phi_q \wedge r_B \wedge \phi'_p,$$

where by ϕ'_p we mean the representation of the state p by ϕ_p (as we defined above) using the primed version of the state and port variables.

The transition relation is encoded by the disjunction of the symbolic encodings of all individual transitions.

Based on the above representation of each ABAR model using Boolean formulas (or switching functions), we can transform the Boolean formulas corresponding to each ABAR into BDDs.

Example 7.1 Figure 7.3 depicts a synchronous channel and its corresponding ABAR model. In this example, we assume that the data set is $\mathcal{D} = \{d_1, d_2\}$). Because there is only one state, we can ignore considering any state variable. Thus, we can represent the model by the following set of Boolean expressions:

$$\begin{gathered}
\mathcal{N}_{Sync} = A \vee B \\
Q_{Sync} = A \wedge B \\
Q_{0_{Sync}} = A \wedge B \\
F_{Sync} = A \wedge B \\
T_{Sync} = (A \wedge B \wedge com_A \wedge com_B \wedge (A = d_1) \wedge (B = d_1) \wedge A' \wedge B') \\
\vee \\
(A \wedge B \wedge com_A \wedge com_B \wedge (A = d_2) \wedge (B = d_2) \wedge A' \wedge B')
\end{gathered}$$

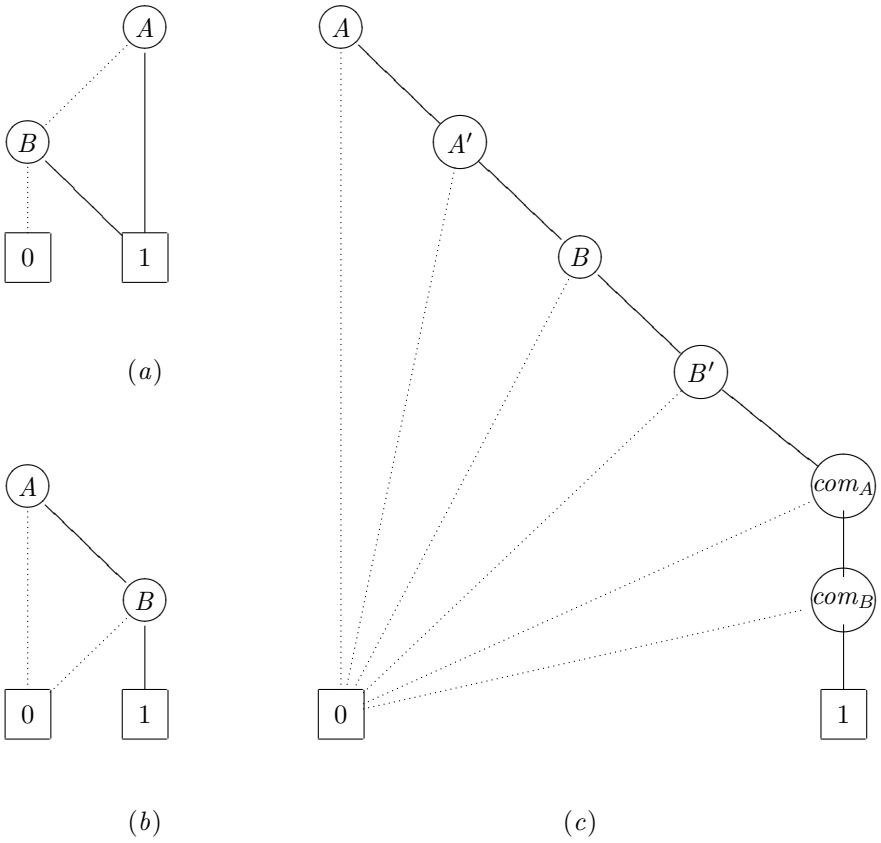


Figure 7.4: BDD representation of a synchronous channel: (a) ports, (b) states, initial states, final states and (c) transition relation.

If we assume that the data set is a singleton $D = \{d\}$, we can abstract away from the data set and the complexities introduced by its multiplicity. As a result the transition relation of the aforementioned synchronous channel can be expressed as follows:

$$T_{Sync} = A \wedge B \wedge com_A \wedge com_B \wedge A' \wedge B'$$

where com_x evaluates to true if and only if port x participates in the corresponding data communication. As the data set is assumed to be a singleton, the set of com_x variables is enough to represent the data communications. In other words, if a variable com_x evaluates to true it means that in its corresponding record we have $x = d$.

Based on the above representation of the ABAR model by Boolean formulas, now we can transform the Boolean formulas corresponding to each ABAR into BDDs. Figure 7.4 depicts a symbolic representation of the Boolean formulas representing the synchronous channel with BDDs. According to Figure 7.3 the single state of the ABAR representing a synchronous

channel is a final state and also an initial state. Therefore, the BDD in Figure 7.4.(a) is enough to store these three pieces of information.

Example 7.2 In a similar way we can symbolically represent a FIFO1 channel. Figure 7.5 depicts a FIFO1 channel, its ABAR interpretation, and a symbolic representation of this ABAR. Because none of the labels of the states implies the other, we use the labels of states as their Boolean representations. We assume that the data set is a singleton $D = \{d\}$. The BDDs in Figure 7.5 are equivalent to the following Boolean expressions:

$$\begin{gathered}\mathcal{N}_{FIFO} = B \vee C \\ Q_{FIFO} = B \vee C \\ T_{FIFO} = (B \wedge com_B \wedge \neg com_C \wedge C') \vee (C \wedge com_C \wedge \neg com_B \wedge B') \\ Q_{0_{FIFO}} = B \\ F_{FIFO} = B\end{gathered}$$

7.2.1 Symbolic Join

So we introduced a symbolic representation for the ABAR models of individual channels. Next, we represent a symbolic equivalent for the join operation of ABAR models. Informally speaking, the join of two ABAR models is computed following two different scenarios. Independent transitions of each ABAR can be interleaved. On the other hand, transitions of two ABARs whose record labels are compatible are synchronized.

Let B_1 and B_2 be two ABARs encoded by $E_{B_1} = \langle \mathcal{N}_1, Q_{B_1}, T_{B_1}, Q_{B_{01}}, F_{B_1} \rangle$ and $E_{B_2} = \langle \mathcal{N}_2, Q_{B_2}, T_{B_2}, Q_{B_{02}}, F_{B_2} \rangle$, respectively. The join of B_1 and B_2 is a generalized augmented Büchi automaton of records B which can be encoded by $E_B = \langle \mathcal{N}, Q_B, T_B, Q_{B_0}, F_B \rangle$ according to following expressions where $\neg com(\mathcal{N}_i)$ stands for $\bigwedge_{n \in \mathcal{N}_i} (\neg com_n)$.

$$\begin{gathered}\mathcal{N} = \mathcal{N}_1 \vee \mathcal{N}_2 \\ Q_B = Q_{B_1} \wedge Q_{B_2} \\ Q_{B_0} = Q_{B_{01}} \wedge Q_{B_{02}} \\ F_B = (F_{B_1} \wedge Q_{B_2}) \wedge (Q_{B_1} \wedge F_{B_2}) \\ T_B = Interleave_1 \vee Interleave_2 \vee Sync_{1,2} \\ Interleave_1 = \bigvee_{q \in Q_2} (\phi_q \wedge T_{B_1} \wedge \neg com(\mathcal{N}_2) \wedge \phi'_q) \\ Interleave_2 = \bigvee_{q \in Q_1} (\phi_q \wedge T_{B_2} \wedge \neg com(\mathcal{N}_1) \wedge \phi'_q) \\ Sync_{1,2} = T_{B_1} \wedge T_{B_2}\end{gathered}$$

Example 7.3 Consider the synchronous channel in Figure 7.3 and the FIFO1 channel in Figure 7.5(a). The join of these two channels is the connector in Figure 7.6(a) with its ABAR model depicted in Figure 7.6(b). The symbolic representation of this connector involves the BDDs in Figures 7.6(c)-(e).

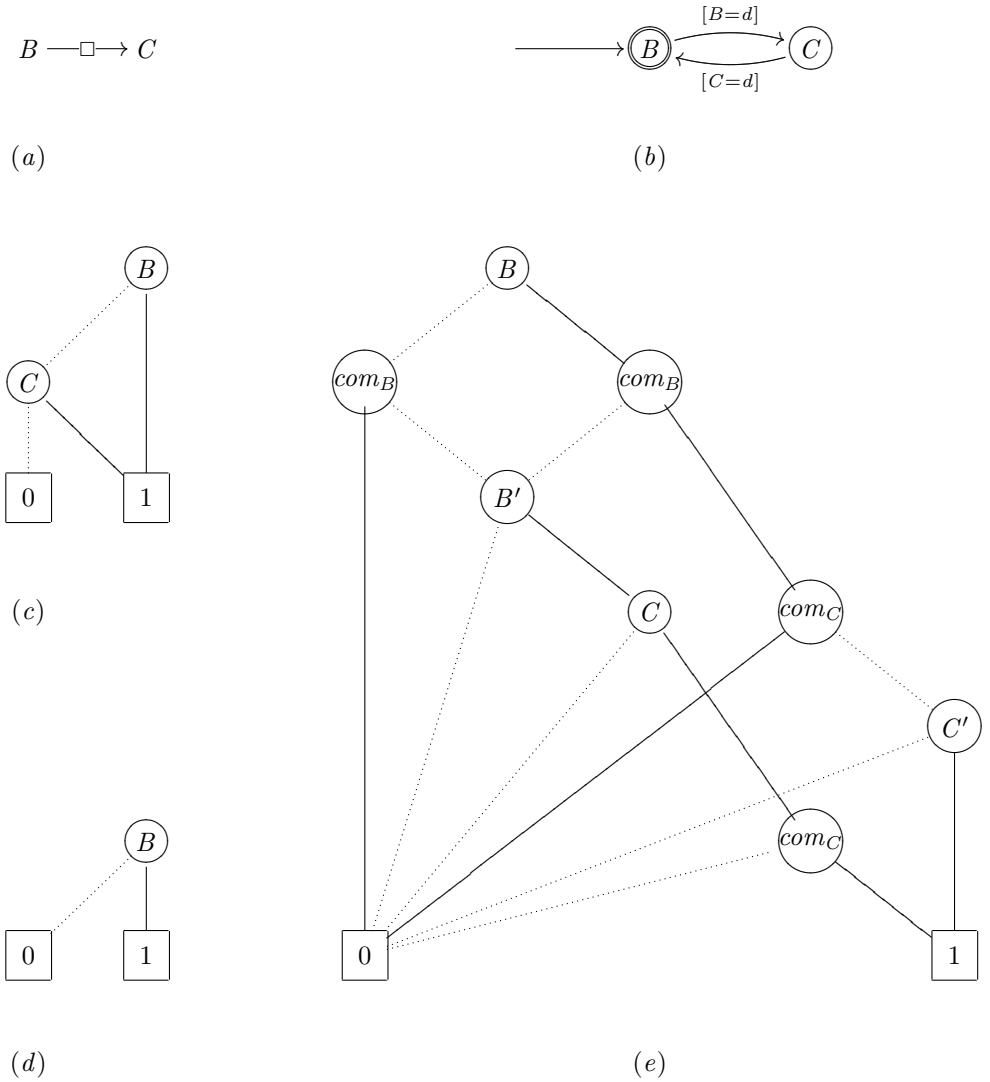


Figure 7.5: (a) FIFO1 channel, (b) its ABAR model, and BDD representation of (c) ports and states, (d) initial states and final states and (e) transition relation.

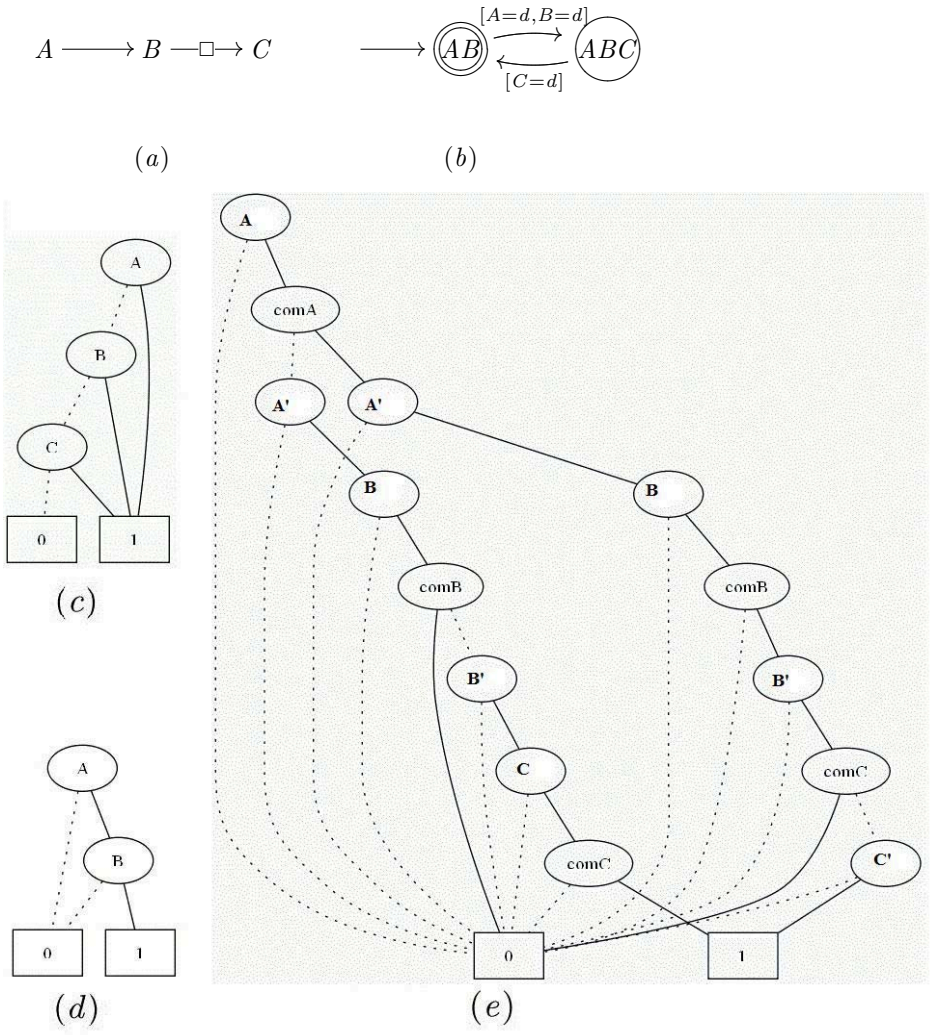


Figure 7.6: (a) Join of a synchronous channel and a FIFO1 channel, (b) its ABAR and BDD representation of (c) ports, (d) states, initial states, final states and (e) transition relation.

7.3 Property Specification by BDD

In the previous section we introduced a symbolic representation for the ABARs corresponding to Reo channels and an operation to mimic the join operation on ABARs in the BDD domain. In this section we continue our symbolic approach by introducing a BDD representation for ρLTL formulas. As mentioned earlier, ρLTL formulas are inductively generated by the following grammar:

$$\phi ::= N \mid \neg\phi \mid \phi \vee \phi \mid \langle r \rangle \phi \mid \phi U \phi.$$

where N is a subset of port names $\mathcal{N}$. Assuming a data set $\mathcal{D}$, r is a record from $Rec_{\mathcal{N}}(\mathcal{D})$.

Our ultimate goal is to implement our methods of global and on-the-fly translations of ρLTL formulas into augmented Büchi automata of records. However, as the first attempt, and in order to use the previously implemented tools (such as LTL2BA [58]) which translate LTL into Büchi automata, we transform ρLTL to LTL. For this purpose, we consider the set of port names $\mathcal{N}$ as a set of atomic propositions. Each $N \subseteq \mathcal{N}$ considered as a ρLTL formula represented by the following LTL formula:

$$\alpha_N = (\bigwedge_{n_i \in N} n_i) \wedge (\bigwedge_{n_i \in \mathcal{N} \setminus N} \neg n_i).$$

Assuming $r = [n_1 = d, n_2 = d, \ldots, n_m = d]$, the ρLTL formula $\langle r \rangle \phi$ is transformed into an LTL formula:

$$\langle r \rangle \phi = r_B \wedge \bigcirc \phi$$

where r_B was defined Section 7.2. As a result, for each ρLTL formula we have a semantically equivalent LTL formula.

Applying the procedure introduced in [58] and implemented as a tool (LTL2BA), an LTL formula is converted to a Büchi automaton whose transition labels are propositional expressions constructed from atomic propositions of the form n and com_n, where $n \in \mathcal{N}$. To unify our approach we transform the result to its equivalent ABAR. Formally, given a Büchi automaton $B = \langle Q, \Sigma, \longrightarrow, Q_0, F \rangle$ where Σ is the set of propositional expressions over the set of atoms $\mathcal{N} \cup \{com_n \mid n \in \mathcal{N}\}$, its equivalent ABAR is $\langle B', l \rangle$ such that the transition labels expressing a data communication constraint are resolved as their equivalent record representation, and the transition labels representing the port enabledness are resolved as the state label of their source state.

Example 7.4 Consider the ρLTL formula $\langle r \rangle (A \wedge B)$ where $r = [A = d, B = d]$. Assuming $\mathcal{D} = \{d\}$, the LTL equivalent of this formula is $com_A \wedge com_B \wedge \bigcirc(A \wedge B)$. A Büchi automaton for this LTL formula is depicted in Figure 7.7(a). This automaton is equivalent to the ABAR depicted in Figure 7.7(b) whit $r = [A = d, B = d]$. In this figure, by using Σ as a transition label, we mean that this transition is enabled for all records in $\Sigma = Rec_{\mathcal{N}}(\mathcal{D})$. Finally, the ABAR in Figure 7.7(b) can be symbolically represented as Figure 7.8. Variables $q1$ and $q2$ are used to encode states. For the case of a transition, $q1$ and $q2$ are used to encode the starting states and $q3$ and $q4$ to encode the target states (respectively, as primed version of $q1$).

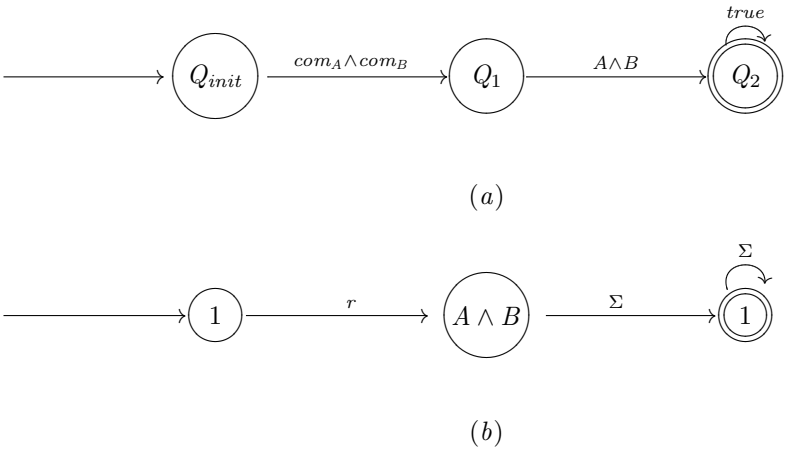


Figure 7.7: (a) A Büchi automaton and (b) an ABAR for $\langle r \rangle (A \wedge B)$

One of our planned future work to enhance our tool is to implement our own on-the-fly translation of ρLTL formulas directly into ABARs.

7.4 A symbolic model checking algorithm

We have introduced a symbolic representation for the ABARs representing Reo connectors in Section 7.2 and a BDD representation for ρLTL formulas in Section 7.3. Next we introduce the algorithm we implemented for model checking of Reo connectors. We follow an automata-based model checking approach. First we assume that the behavioral aspects of a system are modeled by an ABAR model. Second, a system property must be specified by a linear temporal logic. The negation of the property specified by a ρLTL formula is translated into an equivalent automaton. The join of the two automata representing the system and the ρLTL formula is computed. The final stage involves a procedure for language emptiness checking. Emptiness of the language of the resulting automaton implies that the system satisfies the property. Otherwise, the property is violated by the system and any word in the language of the resulting automaton is a counter-example.

Let $B = \langle Q, Rec_{\mathcal{N}}(\mathcal{D}), \longrightarrow, Q_0, F, l \rangle$ be an ABAR model representing the behaviors exhibited by a Reo connector and $B_{\neg\phi} = \langle Q', Rec_{\mathcal{N}}(\mathcal{D}), \longrightarrow', Q'_0, F', l' \rangle$ be an ABAR translation of the negation of a ρLTL formula ϕ. The join of these two ABARs is a generalized ABAR. According to [29] the model checking problem for the formula ϕ is reduced to the classical problem of *fair cycle detection*. Stating the problem in our Büchi context, given the join ABAR $B \bowtie B_{\neg\phi} = \langle Q_{\bowtie}, Rec_{\mathcal{N}}(\mathcal{D}), \longrightarrow_{\bowtie}, Q_{0_{\bowtie}}, \mathcal{F}_{\bowtie} \rangle$ the system violates the property ϕ if and only if there exists a reachable cycle from one of the initial states $q_0 \in Q_0$ such that this cycle is fair with respect to the final states sets of the ABAR $B \bowtie B_{\neg\phi}$.

Not only the problem of LTL model checking, but also some other problems such as

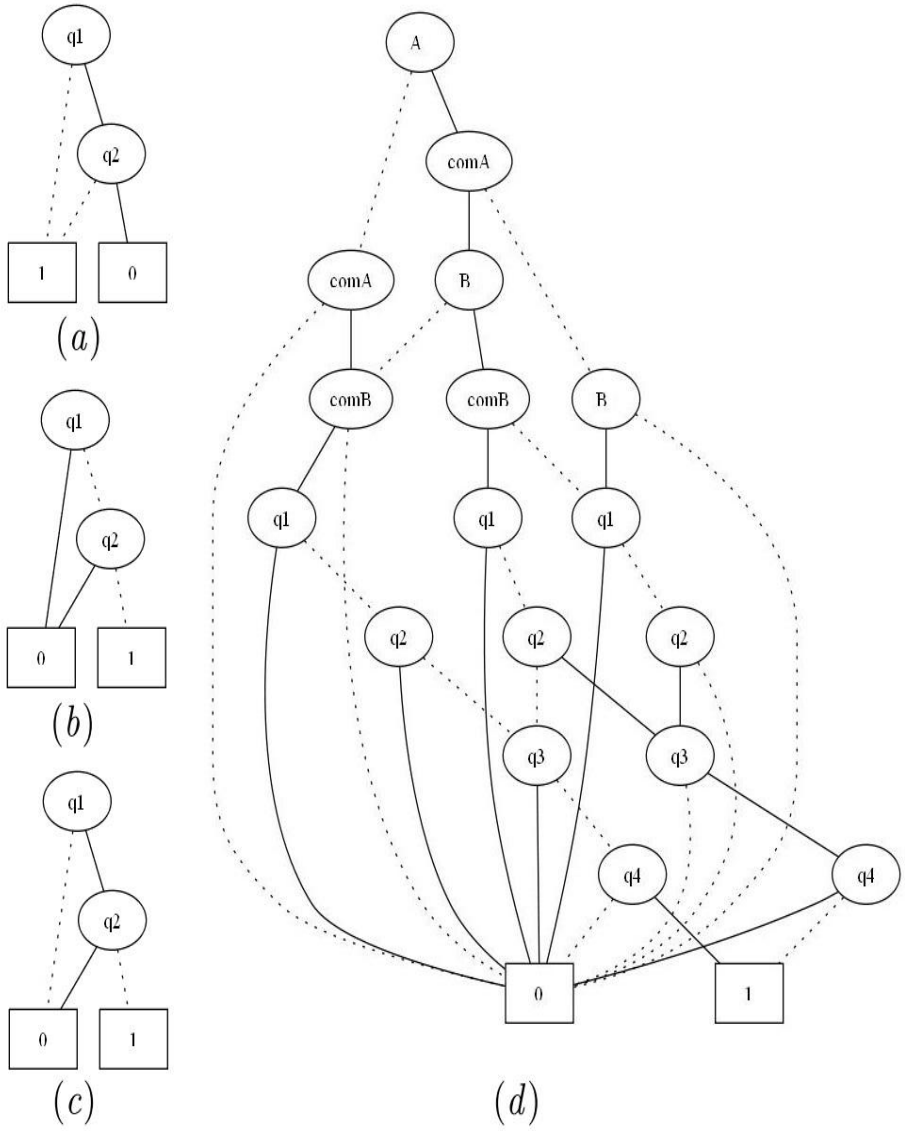


Figure 7.8: BDD representation for the ABAR equivalent of $\langle r \rangle (A \wedge B)$ (a) states, (b) initial states, (c) final states, and (d) transition relation.

language containment based on several types of automata, and CTL model checking with fairness constraints, all reduce to checking the emptiness of the language of a Büchi automaton. The language of the Büchi automaton is nonempty if and only if the automaton contains a *fair cycle*: a (reachable) cycle that contains at least one state from every accepting set, or, equivalently, a *fair strongly connected component* (SCC): a (reachable) nontrivial strongly connected component that intersects each accepting set.

The traditional approach to determine the existence of a fair SCC is to use Tarjans algorithm [137]. This algorithm is based on depth-first search and runs in linear time in the size of the graph. In order to do the depth-first search, the algorithm manipulates the states of the graph explicitly. Unfortunately, as the number of state variables grows, an algorithm that considers every state individually quickly becomes infeasible. Symbolic algorithms [111] manipulate sets of states via their characteristic functions. They derive their efficiency from the fact that in many cases of interest large sets can be described compactly by their characteristic functions. In contrast to explicit algorithms, an advantage of symbolic algorithms, which typically rely on breadth-first search, is that the difficulty of a search is not tightly related to the size of the state space, but is more closely related to the diameter of the graph and the size of the symbolic representation.

Several symbolic algorithms have been proposed that use breadth-first search and compute a set of states that contains all the fair SCCs, without enumerating them [126]. In this case, the typical and standard approach to fair cycle detection is the one of Emerson and Lei [57]. In the last decade, variants of this algorithm and an alternative method based on strongly connected component decomposition have been proposed. In [126], Ravi et al have presented a taxonomy of these techniques using some fix-point logic representations and compare representatives of each major class on a collection of real-life examples. Their main result indicates that the Emerson-Lei procedure is the fastest, but other algorithms tend to generate shorter counter-examples [126]. Based on this result, the algorithm implemented in our model checking tool is the one of Emerson and Lei [57].

7.5 Experimental results

In this section we present some experimental results applying our implementation based on the concepts introduced in the previous sections. Our model checking tool is implemented in C++ and compiled with GCC. The reported results are achieved on a Pentium 4, 2.53 GHz, 4GB RAM with an Ubuntu operating system. We use JINC [3] binary decision diagram library in our tool. In the following case studies, for the Reo channels we suppose that the data set is $\{d\}$ and we apply the ABAR models depicted in Figures 7.9.

7.5.1 Dining philosophers

The classical dining philosophers problem can be described as a coordination system by Reo specifications [14]. This system can be designed as a set of pairs of instances of two components: philosopher and chopstick instances. The externally observable behavior of a

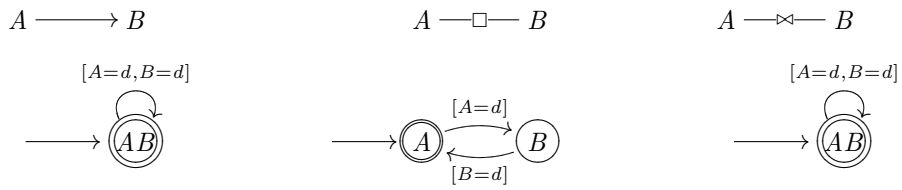


Figure 7.9: ABAR models of some Reo channels where $\mathcal{D} = \{d\}$.

Table 7.1: State space generation results for the dining philosophers problem

n	Time(sec)	Number of generated states	Number of BDD nodes for transition relation
2	0.236	36	245
3	0.533	216	473
4	3.810	1296	1043
5	92.515	7776	1956

philosopher component is as follows. After some period of thinking, it decides to eat, attempts to obtain its two chopsticks by issuing requests on its TL_i and TR_i ports. We assume that it always issues a request through its left chopstick before requesting the one on its right. Once both chopsticks take requests are granted it proceeds to eat for some time, at the end of which the philosopher then issues requests to free its left and right chopsticks by writing tokens on its RL_i and RR_i ports.

A chopstick is modeled by a FIFO1 channel and a synchronous drain [14]. The coordination pattern for this problem is represented in Figure 7.10 for the special case where the number of philosophers is 2. The coordination scenario and its graphical representation can simply be expanded for more than two philosophers [14]. Considering the philosophers as the active components in our system that communicate through this network, the behavioral aspects of such components can be modeled as Figure 7.11. In fact, q_1, q_2 and q_3 are respectively the $thinking$, $waiting$ and $eating$ states of each philosopher.

In Table 7.1, we see the number of states and the BDD nodes for different numbers of philosophers. For each case the state space generation time, number of generate states in the corresponding ABAR, and the number of BDD nodes in the symbolic representation of the transition relation (as it dominates other parts of our encoding considering the space complexity) are illustrated.

Let ϕ_1 and ϕ_2 be two ρLTL formulas in the context of dining philosophers problem such that:

$$\phi_1 = \Box\neg(eating_1 \wedge eating_2)$$
$$\phi_2 = \Box(TR_i \rightarrow \langle TR_i\rangle(RL_i \wedge RR_i)).$$

The property ϕ_1 asserts that in all instance of time, it is not the case that both philosophers are eating simultaneously. For the case of more than two philosophers, the inner conjunction in the formula ϕ_1 is expanded by all $eating_i$'s. The property ϕ_2 that is completely specified in terms of the port names, says that always if the philosopher i is waiting to take the right

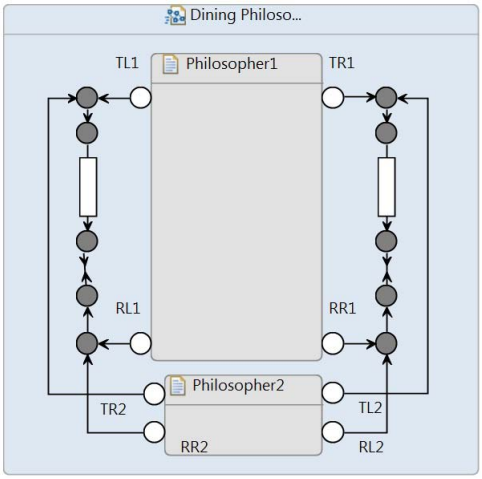


Figure 7.10: Coordination pattern for two philosophers in the dining philosophers problem

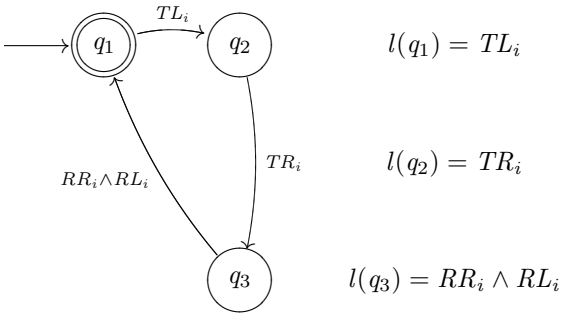


Figure 7.11: Behavior of a philosopher in ABAR terms

chopstick, it immediately takes it and goes to the eating state. obviously, the first property, ϕ_1, is satisfied by the presented coordination scenario but the second one, ϕ_2, is not satisfied since the readiness of a philosopher for taking its right chopstick does not lead to the $eating$ state for it exactly in the next state of the whole system.

Table 7.2 reports the model checking time for various number of philosophers for properties ϕ_1 and ϕ_2. For model checking the formula ϕ_1, the algorithm terminates after checking the whole state space, while for the case of ϕ_2, it terminates after finding a counterexample.

7.5.2 Mutual Exclusion

As another case study we consider a special variant of the mutual exclusion problem. Let n be the number of active processes in a system. Assuming $k \leq n$, in this variant of the mutual exclusion problem, at each time instance at most k processes can perform actions in

Table 7.2: Model checking time (sec) for n dining philosophers

n	ϕ_1	ϕ_2
2	0.260	0.258
3	0.611	0.603
4	4.246	4.272
5	96.098	96.009

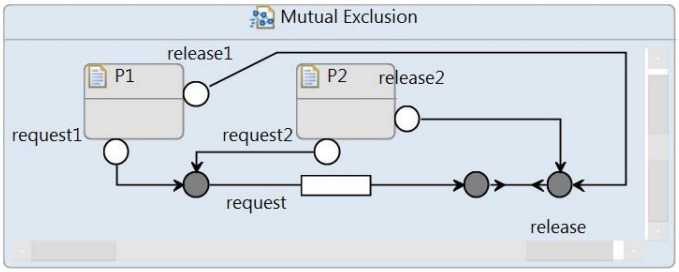


Figure 7.12: Coordination pattern for two processes in mutual exclusion for $k = 1$

the critical section. Figure 7.12 presents the communication pattern for this problem for a special case where $n = 2$ and $k = 1$.

The active processes in the system are considered as components that communicate through this connector. Figure 7.13 models the behavior of a process with an ABAR. Table 7.3 reports some results on state space generation in mutual exclusion problem for different numbers k of processes in the critical section, and active processes n, ($k \leq n$). For model checking purposes the labeling function of the ABAR depicted in Figure 7.13 must be extended to show the state of a process (executing critical/non-critical actions). Obviously, the state labeled by $release_i$ represents a configuration where a process executes actions in the critical sections. Let ϕ_3 and ϕ_4 be two ρLTL formulas such that:

$$\phi_3 = \Box\neg(critical_1 \wedge critical_2 \wedge critical_3)$$
$$\phi_4 = \Box(request_1 \rightarrow \langle request_1\rangle(\neg request_1))$$

Property ϕ_3 specifies that three processes cannot execute their critical actions simultaneously, a property which is true as long as $k<3$. Property ϕ_4 expresses that for a process ready to enter its critical section, always the next action is a request to execute its critical actions and this request is always accepted. This property is not always true.

Table 7.4 represents the model checking time for properties ϕ_3 and ϕ_4 for various values of n and k.

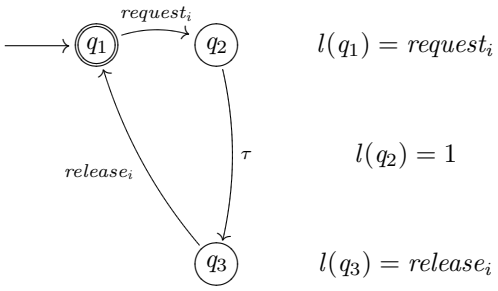


Figure 7.13: Behavior of a process in ABAR terms

Table 7.3: State space generation results for the mutual exclusion problem

n	k	Time(sec)	Number of generated states	Number of BDD nodes for transition relation
5	2	0.378	972	4421
6	2	0.868	2916	9830
7	2	1.134	8748	21767
8	2	3.105	26244	47912
9	2	30.318	236196	227690
10	3	60.090	472392	532209

7.5.3 Discussion

For model checking of the desired properties of coordination models specified by Reo, in addition to our above mentioned tool, there is another implemented tool called Vereofy [7]. The two above introduced case studies have also been considered by the authors of Vereofy and their results have been reported in [98, 99]. Similar to our implementation, they use OBDD as their main data structure to store and process their models. However, there are some essential differences between our approach to model checking of Reo nets and the work of Baier et al reported in [98, 99]:

- The modeling formalism that they use is constraint automaton while our models are

Table 7.4: Model checking time (sec) for the mutual exclusion problem

n	k	ϕ_3	ϕ_4
5	2	0.539	0.532
6	2	1.270	1.252
7	2	3.283	3.425
8	2	10.460	10.887
10	2	105.115	107.285

Büchi automata of records and their augmented versions.

- The property specification language that they use is an extension of the *branching time* temporal logic CTL called *BTSL* while our proposed logic is an action based *linear time* temporal logic called ρLTL.
- The algorithm of model checking used in the work of Baier et al is an extension of the symbolic CTL model checking algorithm [48] which by an iterative approach that computes the set of states satisfying each subformula of the desired property. Our model checking algorithm is based on the checking of the emptiness of the accepted language of an automaton that is reduced to the problem of detecting fair cycles in the graphs of automata.

8 Compositional Reduction

In the previous chapters, we introduced constraint automata, Büchi automata of records and their augmented versions as operational models for Reo connectors. We have shown that they have increasing expressiveness. We also introduced methods for model checking of Reo nets using both global and on-the-fly translations of linear temporal logic formulas into automata. Now, we deal with the problem of *state explosion*, namely that the model of the systems tend to be extremely large. In this chapter we investigate the method of *compositional reduction* to deal with the problem of state explosion for the case of large scale Reo nets. We concentrate on the most basic semantic model of Reo, namely constraint automata, and we leave for future work the investigation of similar compositional reduction techniques for ABAR models. In Section 8.1, we introduce the method and overview the way in which we are able to minimize the models of Reo nets. In the subsequent sections, we present the technical details with some examples.

8.1 Introduction

Equivalence based *compositional reduction* is a way to deal with the problem of state explosion [50, 139]. In this method, the models of the components of a system are reduced with respect to an *equivalence relation* before building the model of the whole system [60, 50, 79, 82]. In order to be useful, the equivalence relation should satisfy two properties: *preservation* of all properties to be verified and being a *congruence* relation with respect to all operators that are used for composing the models. By a congruence relation we mean that the replacement of a component of a model by an equivalent one should always yield a model that is equivalent with the original one.

When transition systems are used as the semantics of specification formalisms, one of the key questions is whether two models are equivalent. In the case of labeled transition systems with simple alphabets, numerous equivalence relations have been presented in the literature. *Trace equivalence*, *visible-trace equivalence* (automata-theoretic equivalence), weak and strong *bisimilarity* presented by Milner [112], *failure-based equivalences*, and CSP-like equivalences presented by Hoare [62] are examples of these equivalences. (For a survey on several equivalence relations see [143, 144].) From a theoretical point of view, the investigation of these equivalences in the case of labeled transition systems with compound alphabets such as constraint automata and record-based labeled transition systems are interesting.

Fortunately, in the context of failure based semantic models of the process description language LOTOS, there are two equivalence relations, called *chaos-free failures divergences (CFFD)* and *non-divergent failure divergences (NDFD)*, which satisfy the preservation property for two fragments of linear temporal logic. NDFD preserves linear time temporal logic without next-time operator (called LTL_{-X}) [86]. CFFD preserves linear temporal logic without the next-time operator but with an extra operator that distinguishes deadlocks from divergences (called LTL^{ω}) [141, 142]. Also, it has been shown that CFFD and NDFD are the weakest equivalence relations that preserve the above mentioned fragments of linear temporal logic [86, 141]. In addition, it has been shown that in the case of labeled transition systems with simple alphabets, CFFD and NDFD are congruences with respect to all composition op-

erators defined in LOTOS [142].

Now, we investigate the above mentioned results for the case of constraint automata. In other words, instead of their TDS-based semantics, we consider the failure based semantics for constraint automata as labeled transition systems with compound labels. Thus, first we define CFFD and NDFD equivalences for constraint automata. Then, we show that the temporal logic preservation results also will hold in these cases. Next, we consider the congruency results. Obviously, if we consider constraint automata as labeled transition systems with the composition operators defined in LOTOS, then the previously established congruency results for CFFD and NDFD also hold for constraint automata. In this chapter, we consider two other composition operators that refer to the internal structures of the transition labels. These two composition operators are the operators of join and hiding a port name, as we introduced them in Chapter 3. We prove that failure-based equivalence relations CFFD and NDFD are congruences with respect to both join and hiding operators of constraint automata. Therefore, based on the congruency results, and because of the linear time temporal logic preservation properties of CFFD and NDFD equivalences and their minimality properties, CFFD and NDFD can be used for compositional reduction of constraint automata models in the field of model checking.

8.2 Failure based equivalence of constraint automata

Now, we define the notions of CFFD and NDFD-equivalence relations. We define these equivalences for *labeled transition systems* in general and for constraint automata in particular. First, recall the notion of labeled transition systems:

Definition 8.1
- A *transition alphabet* is a countable set of symbols Σ not containing the empty transition label τ.
- We write Σ_τ for $\Sigma \cup \{\tau\}$, and Σ^* (Σ^ω) for the set of all finite (infinite) words consisting of elements of Σ. The symbol τ is used to denote the empty word.
- If $\sigma \in (\Sigma_\tau^* \cup \Sigma_\tau^\omega)$, $vis(\sigma)$ is used to denote the word obtained by removing all τ-symbols from σ and $\Sigma(\sigma)$ denote the set of elements of σ.
- A *labeled transition system (LTS)* is a triple $L = \langle S, s, \Delta \rangle$, where S is the set of states, $s \in S$ is the initial state and $\Delta \subseteq S \times \Sigma_\tau \times S$ is the transition relation.
- The alphabet of L , $\Sigma(L)$ is the set: $\Sigma(L) = \{l \in \Sigma |\ \exists s, s' : (s, l, s') \in \Delta\}$. The alphabet of any *LTS* is required to be finite. In addition, an LTS is finite if its set of states is finite.

Now we introduce some operators that can be used to compose labeled transition systems. These operators are parallel composition with the possibility of synchronization on some transition labels, nondeterministic choice, simple hiding, and renaming.

Definition 8.2 Let $L_1 = \langle S_1, s_1, \Delta_1 \rangle$ and $L_2 = \langle S_2, s_2, \Delta_2 \rangle$ be two LTSs.
(i) The *parallel composition* of L_1 and L_2 with respect to $G = \{g_1, \ldots, g_n\} \subseteq \Sigma$, denoted by $L_1 |[g_1, \ldots, g_n]| L_2$, is the LTS $\langle S_1 \times S_2, (s_1, s_2), \Delta \rangle$, where
- $((t, u), g_i, (t', u')) \in \Delta$, for $g_i \in G$, iff $(t, g_i, t') \in \Delta_1$ and $(u, g_i, u') \in \Delta_2$, and

- $((t,u),l,(t',u')) \in \Delta$ for $l \notin G$, iff either $(t,l,t') \in \Delta_1$ and $u = u'$ or $(u,l,u') \in \Delta_2$ and $t = t'$.
(ii) The *nondeterministic choice* composition of L_1 and L_2, denoted by $L_1[\,]L_2$, is the LTS $\langle S_1 \times \{1\} \cup S_2 \times \{2\} \cup \{(s,0)\}, (s,0), \Delta\rangle$, where
- $((t,i),l,(t',i)) \in \Delta$, where $i \in \{1,2\}$, iff $(t,l,t') \in \Delta_i$, and
- $((s,0),l,(t,i)) \in \Delta$, where $i \in \{1,2\}$, iff $(s_i,l,t) \in \Delta_i$.

Definition 8.3 Let $L_1 = \langle S_1, s_1, \Delta_1\rangle$ be an LTS and $G = \{g_1,\ldots,g_n\} \subset \Sigma$ and $H = \{h_1,\ldots,h_n\} \subset \Sigma$.
(i) The *simple hiding* of G in L_1, denoted by $Hide\ g_1,\ldots,g_n\ in\ L_1$, is the LTS $\langle S_1, s_1, \Delta\rangle$ where
- $(t,l,t') \in \Delta$, iff either $l \notin G$ and $(t,l,t') \in \Delta_1$ or $l = \tau$ and there is a $g_i \in G$ such that $(t,g_i,t') \in \Delta_1$.
(ii) The *renaming* of L_1 with respect to G and H, denoted by $L_1[h_1/g_1,\ldots,h_n/g_n]$, is the LTS $\langle S_1, s_1, \Delta\rangle$ where
- $(t,l,t') \in \Delta$ iff either $l \notin G$ and $(t,l,t') \in \Delta_1$ or $l = h_i$ and $(t,g_i,t') \in \Delta_1$.

Now, we recall some basic concepts of process algebra and give the definitions of CFFD and NDFD-equivalences [141, 142, 86].

Definition 8.4 Let $L = \langle S, s, \Delta\rangle$ be a labeled transition system.
- If $\rho \in \Sigma_\tau^*$, we write $s_0 \xrightarrow{\rho} s_n$ for $n = |\rho|$ iff there are $s_1,...,s_{n-1}$ such that for all $0 < i \leq n$, $(s_{i-1},\rho_i,s_i) \in \Delta$.
- If there is an s_n such that $s_0 \xrightarrow{\rho} s_n$ we write $s_0 \xrightarrow{\rho}$.
- If $\rho \in \Sigma_\tau^\omega$, we write $s_0 \xrightarrow{\rho}$ iff $\exists s_1, s_2, ...$ such that for all $i > 0$, $(s_{i-1},\rho_i,s_i) \in \Delta$.
- If $\sigma \in (\Sigma^* \cup \Sigma^\omega)$, we write $s_0 \stackrel{\sigma}{\Longrightarrow} s_n$ ($s_0 \stackrel{\sigma}{\Longrightarrow}$) iff there is a $\rho \in (\Sigma_\tau^* \cup \Sigma_\tau^\omega)$ such that $s_0 \xrightarrow{\rho} s_n$, ($s_0 \xrightarrow{\rho}$) and $\sigma = vis(\rho)$.

Now, we can define the notions of traces, divergence, stability and failures for labeled transition systems in general, based on [86]:

Definition 8.5 Let $L = \langle S, s, \Delta)\rangle$ be a labeled transition system.
- $\sigma \in \Sigma^*$ is a *trace* of L iff $s \stackrel{\sigma}{\Longrightarrow}$.
- $tr(L)$ is the set of all traces of L.
- $\sigma \in \Sigma^\omega$ is an *infinite trace* of L iff $s \stackrel{\sigma}{\Longrightarrow}$.
- $inftr(L)$ is the set of all infinite traces of L.
- $\sigma \in \Sigma^*$ is a *divergence trace* of L iff there is a $\rho \in \Sigma_\tau^\omega$ such that $s \xrightarrow{\rho}$ and $\sigma = vis(\rho)$.
- $divtr(L)$ is the set of all divergence traces of L.
- $s' \in S$ is *stable*, if not $s' \xrightarrow{\tau}$.
- An LTS L is stable if its initial state s is stable. We write $stable(L)$ if L is stable, and $\neg stable(L)$ if it is not.
- $(\sigma, A) \in \Sigma^* \times 2^\Sigma$, where 2^Σ denotes the power set of Σ, is a *failure* of L iff there is an $s' \in S$ such that $s \stackrel{\sigma}{\Longrightarrow} s'$ and $\forall a \in A.\neg(s' \stackrel{a}{\Longrightarrow})$.
- $fail(L)$ is the set of all failures of L.
- $(\sigma, A) \in \Sigma^* \times 2^\Sigma$ is a *stable failure* of L iff there is a stable $s' \in S$ such that $s \stackrel{\sigma}{\Longrightarrow} s' \wedge \forall a \in A.\neg(s' \stackrel{a}{\Longrightarrow})$.

- $sfail(L)$ is the set of all stable failures of L.
- $(\sigma, A) \in \Sigma^* \times 2^\Sigma$ is a *divergence-masked failure* of L iff (σ, A) is a failure or σ is a divergence trace.
- $dfail(L)$ is the set of divergence-masked failures of L.

The following lemma lists some direct consequences of the above definition for later use.

Lemma 8.1 Let L be a labeled transition system,
a) $tr(L) = divtr(L) \cup \{\sigma | (\sigma, \emptyset) \in sfail(L)\}$.
b) $tr(L) = \{\sigma | (\sigma, \emptyset) \in fail(L)\} = \{\sigma | (\sigma, \emptyset) \in dfail(L)\}$.
c) $dfail(L) = sfail(L) \cup (divtr(L) \times 2^\Sigma)$.
d) If L is a finite labeled transition system,
$$inftr(L) = \{\omega \in \Sigma^\omega | \forall \sigma \in \Sigma^* : (\sigma \textit{ is a proper prefix of } \omega \rightarrow \sigma \in tr(L))\}.$$

Now, we introduce two failure based equivalences for labeled transition systems that were originally introduced in [141, 142]:

Definition 8.6 Let L and L' be two labeled transition systems.
(i) We say that L and L' are CFFD equivalent and write $L \overset{cffd}{\approx} L'$ if and only if $stable(L) \Leftrightarrow stable(L')$, $divtr(L) = divtr(L')$, $inftr(L) = inftr(L')$ and $sfail(L) = sfail(L')$.
(ii) We say that L and L' are NDFD equivalent and write $L \overset{ndfd}{\approx} L'$ if and only if $stable(L) \Leftrightarrow stable(L')$, $divtr(L) = divtr(L')$, $inftr(L) = inftr(L')$ and $dfail(L) = dfail(L')$.

The NDFD-equivalence is strictly weaker than CFFD-equivalence in the sense of the following lemma:

Lemma 8.2 If $L \overset{cffd}{\approx} L'$, then $L \overset{ndfd}{\approx} L'$.

If the labeled transition systems examined are finite, the component $inftr$ in the above definitions is superfluous. Now, we define the notion of being *congruence* for equivalence relations with respect to a composition operator:

Definition 8.7 Let $\approx$ be an equivalence relation and f be a composition operator over a set of labeled transition systems. We say that $\approx$ is a *congruence* with respect to f iff for every $L_1, \ldots, L_n$ and $L'_1, \ldots, L'_n$ such that $L_i \approx L'_i$ the following holds: $f(L_1, \ldots, L_n) \approx f(L'_1, \ldots, L'_n)$.

Obviously, each constraint automaton $C = \langle Q, \mathcal{N}, \rightarrow, q_0 \rangle$ over data set $\mathcal{D}$ can be considered as a labeled transition system with alphabet

$$\Sigma = \{(N, g) | N \subseteq \mathcal{N} \wedge g \in DC(\mathcal{N}, \mathcal{D}) \wedge N \neq \emptyset\}:$$

Lemma 8.3 For a constraint automaton $C = \langle Q, \mathcal{N}, \rightarrow, q_0 \rangle$ over a data set $\mathcal{D}$, let $L(C) = (S, s, \Delta)$ be the labeled transition system over the alphabet $\Sigma = \{(N, g) | N \subseteq \mathcal{N} \wedge g \in DC(\mathcal{N}, \mathcal{D}) \wedge N \neq \emptyset\}$, where, $S = Q$, $s = q_0$ and $(q_i, (N, g), q_j) \in \Delta$ if and only if $(q_i, N, g, q_j) \in \rightarrow$. Then, the constraint automata C and C' are (TDS-based) equivalent if and only if they are infinite-trace-based equivalent. In other words $L_{TDS}(C) = L_{TDS}(C')$ if and only if $inftr(L(C)) = inftr(L(C'))$.

Proof. This lemma is a direct consequence of Definitions 3.7 and 8.5 □

Based on the above lemma, if we consider the elements of the alphabet of every constraint automaton as simple elements and do not refer to their internal structures then, constraint automata can be composed using every well defined operator for composing labeled transition systems, such as parallel composition with synchronization, nondeterministic choice, and renaming. In addition, in the Chapter 3 we introduced two composition operators, join and hiding with respect to a port name, whose definitions depend on the internal structures of the elements of the alphabet sets of constraint automata.

In [142] it has been proved that CFFD and NDFD (without the need to check for the stability predicates) are congruences with respect to all basic composition operators of LOTOS, except for the operator of nondeterministic choice. For the case of nondeterministic choice operator, it is also necessary to check the stability predicate. For composing constraint automata, not only we can use these operators, but we also have the two extra operators (join and hiding a port name) which refer to the internal structure of the elements of alphabet sets. In the following sections, we show that equivalence relations CFFD and NDFD are also congruences with respect to both join and hiding operators of constraint automata.

8.3 Congruency Results for Joining of Constraint Automata

In this section, we prove that the equivalence relation CFFD is a congruence with respect to the join of constraint automata and it is also the case for the equivalence relation NDFD. Our method of proof is a modification and extension of the proof of that CFFD and NDFD relations are congruences for the case of parallel composition of LTSs presented in [142].

First, we define a predicate **Join**$(\sigma;\pi,\rho)$, which intuitively means that words π and ρ can be considered as traces of two constraint automata while σ is a trace in the join constraint automaton resulting from the join of ρ and π.

Definition 8.8 Let $Data$ be a set of data, Nam_1 and Nam_2 be sets of names. Let $\Sigma_1 = \{(N,g)|N \subseteq Nam_1 \wedge N \neq \emptyset \wedge\ g \in DC(N,Data)\}$, $\Sigma_2 = \{(N,g)|N \subseteq Nam_2 \wedge N \neq \emptyset \wedge\ g \in DC(N,Data)\}$, $\Sigma = \{(N,g)|N \subseteq Nam_1 \cup Nam_2 \wedge N \neq \emptyset \wedge\ g \in DC(N,Data)\}$ and $\sigma = (N_1,g_1)(N_2,g_2)\ldots$ be a finite or infinite word over the alphabet Σ. We define the predicate **Join**$(\sigma;\pi,\rho)$ to hold (to be true) if and only if there is a function *moved* from $\{1,2,\ldots\}$ to $\{first, second, both\}$ such that:

1-

$$moved(i) = \begin{cases} first & \text{if } N_i \cap Nam_2 = \emptyset \text{ and } g_i \in DC(Nam_1, Data), \\ second & \text{if } N_i \cap Nam_1 = \emptyset \text{ and } g_i \in DC(Nam_2, Data), \\ both & \text{otherwise.} \end{cases}$$

2- π is obtained from σ by:

2-1- for all $i \geq 1$ where, $moved(i) = both$, change (N_i, g_i) to

$(N_i \cap Nam_1, g_i[Nam_1])$,
2-2- remove all (N_i, g_i) where, $moved(i) = second$.
3- ρ is obtained from σ by:
3-1- for all $i \geq 1$ where, $moved(i) = both$, change (N_i, g_i) to
$(N_i \cap Nam_2, g_i[Nam_2])$,
3-2- remove all (N_i, g_i) where, $moved(i) = first$.

By $g[Nam_i]$ we mean the restriction of data constraint g to the name set Nam_i: in the conjunctive normal form of g, the restricted $g[Nam_i]$ can be obtained by replacing all terms containing $d_A = d$ where $A \notin Nam_i$ with $true$. Obviously, the obtained word π is a word over alphabet Σ_1 and ρ is a word over alphabet Σ_2.

Now, we show that the sets of finite or infinite traces, stable failures, divergent traces and divergence-masked failures of the join automaton can be characterized by their counterparts in the two constraint automata. Based on these characterizations, we prove our congruency results.

Proposition 8.4 Let $C_1 = \langle Q_1, Nam_1, T_1, q_{01}\rangle$ and $C_2 = \langle Q_2, Nam_2, T_2, q_{02}\rangle$ be two constraint automata. Then,
(i) $tr(C_1 \bowtie_C C_2) = \{\sigma \mid \exists \pi \in tr(C_1),\ \exists \rho \in tr(C_2),\ Join(\sigma;\pi,\rho)\}$.
(ii) $sfail(C_1 \bowtie_C C_2) = \{(\sigma, A) \mid \exists (\pi, B) \in sfail(C_1),\ \exists (\rho, D) \in sfail(C_2),$
$Join(\sigma;\pi,\rho)\ \ and\ A \cap G \subseteq B \cap D\ \wedge\ A \cap G' \subseteq B \cup D\}$,
where,
$G = \{(N, g) |\ N \subseteq Nam_1 \cup Nam_2 \wedge N \neq \emptyset \wedge (N \cap Nam_1 = \emptyset \vee N \cap Nam_2 = \emptyset)\}$,
$G' = \{(N, g) |\ N \subseteq Nam_1 \cup Nam_2 \wedge N \neq \emptyset \wedge (N \cap Nam_1 \neq \emptyset \wedge N \cap Nam_2 \neq \emptyset)\}$.
(iii) $stable(C_1 \bowtie_C C_2) = stable(C_1) \wedge stable(C_2)$.
(iv) $divtr(C_1 \bowtie_C C_2) = \{\sigma \mid \exists \pi \in tr(C_1),\ \exists \rho \in tr(C_2), Join(\sigma;\pi,\rho)\ and$
$(\pi \in divtr(C_1) \vee \rho \in divtr(C_2))\}$.
(v) $dfail(C_1 \bowtie_C C_2) = \{(\sigma, A) \mid \exists (\pi, B) \in dfail(C_1),\ \exists (\rho, D) \in dfail(C_2),$
$Join(\sigma;\pi,\rho)\ \ and\ A \cap G \subseteq B \cap D\ \wedge\ A \cap G' \subseteq B \cup D\}\ \cup\ (divtr(C_1 \bowtie_C C_2) \times 2^{\Sigma})$,
where, Σ is the same as defined in Definition 8.8 and G and G' are the same as defined in (ii).
(vi) $inftr(C_1 \bowtie_C C_2) = \{\omega \mid \exists \pi \in tr(C_1) \cup inftr(C_1),\ \exists \rho \in tr(C_2) \cup inftr(C_2),$
$Join(\omega;\pi,\rho) \wedge (\pi \in inftr(C_1) \vee \rho \in inftr(C_2))\}$.

Proof.
First note that in general constraint automata can be nondeterministic, i.e. there are transitions with the same source states and the same labels but with different target states. Thus, the last state after a finite trace can be more than one and for a trace σ in the join of two constraint automata the predicate $Join(\sigma;\pi,\rho)$ can be satisfied by more than one pair of traces (π,ρ). Now we prove the proposition:

(i) This proposition is a direct consequence of Definitions 8.5, 3.8 and 8.8.

(ii) Let $(\pi, B) \in sfail(C_1)$, $(\rho, D) \in sfail(C_2)$ and $Join(\sigma;\pi,\rho)$. We prove that for all $A \subseteq \Sigma$, if $A \cap G \subseteq B \cap D\ \wedge\ A \cap G' \subseteq B \cup D$, then $(\sigma, A) \in sfail(C_1 \bowtie_C C_2)$.

First note that, $\pi \in tr(C_1)$, $\rho \in tr(C_2)$ and $Join(\sigma;\pi,\rho)$, thus based on Proposition 8.4(i), $\sigma \in tr(C_1 \bowtie_C C_2)$ and because (π, B) and (ρ, D) are stable failures, there is no outgoing transition with label τ from the last state in $C_1 \bowtie_C C_2$ after tracing σ. We denote this state by q_F, the last state in C_1 after tracing π by q_B and the last state in C_2 after tracing ρ by q_D. Let A be the greatest member of 2^Σ such that $A \cap G \subseteq B \cap D \ \wedge \ A \cap G' \subseteq B \cup D$. (Since Σ is finite, such a set exists). Now using proof by contradiction, suppose that there is an outgoing transition from state q_F in $C_1 \bowtie_C C_2$ with label $(N, g) \in A$. Based on Definition 3.8, we have three cases, based on N: (1) $N \subseteq Nam_1$ and $N \cap Nam_2 = \emptyset$. In this case, $(N, g) \in A \cap G$. Thus, $(N, g) \in B \cap D$. But, both (ρ, D) and (π, B) are fail runs in their corresponding automata. Thus, it is impossible for (N, g) to be the label of an outgoing transition from q_F in the product automaton. (2) $N \subseteq Nam_2$ and $N \cap Nam_1 = \emptyset$. The proof is symmetric with case (1). (3) $N = N_1 \cup N_2$ where $N_1 \subseteq Nam_1$, $N_2 \subseteq Nam_2$ and $N_1 \cap Nam_2 = N_2 \cap Nam_1$. In this case, $(N, g) \in A \cap G'$. Thus, either $(N, g) \in B$ or $(N, g) \in D$. In either case it is impossible for (N, g) to be the label of an outgoing transition from q_F in the product automaton, because at least one of the states q_B and q_D does not have an outgoing transition with label (N, g) in its corresponding automaton. Because we supposed that A is the greatest subset of Σ where $A \cap G \subseteq B \cap D \ \wedge \ A \cap G' \subseteq B \cup D$, our claim holds for the smaller subsets of Σ.
On the other hand, let $(\sigma, A) \in sfail(C_1 \bowtie_C C_2)$. Thus $\sigma \in tr(C_1 \bowtie_C C_2)$ and based on Proposition 8.4(i), there are $\pi \in tr(C_1)$ and $\rho \in tr(C_2)$ such that $Join(\sigma;\pi,\rho)$. Let B be the greatest subset of Σ where $(\pi, B) \in fail(C_1)$ and D be the greatest subset of Σ where $(\rho, D) \in fail(C_2)$. Again, we denote the last state in C_1 after tracing π by q_B, the last state in C_2 after tracing ρ by q_D and the last state in $C_1 \bowtie_C C_2$ after tracing σ by q_F. Because q_F is stable, based on Definition 3.8, q_B and q_D are stable. Thus, (π, B) and (ρ, D) are stable failures. If $(N, g) \in A \cap G$ then $N \cap Nam_1 = \emptyset$ or $N \cap Nam_2 = \emptyset$ and there is no outgoing transition with label (N, g) from q_F. If $N \cap Nam_1 = \emptyset$ then obviously, $(N, g) \in B$ and based on Definition 3.8 it cannot be the label of an outgoing transition from q_D in C_2 . Thus, because of the maximality of D, $(N, g) \in D$. Thus, $(N, g) \in B \cap D$. Similarly, if $N \cap Nam_2 = \emptyset$ then $(N, g) \in B \cap D$. Thus, $A \cap G \subseteq B \cap D$. If $(N, g) \in A \cap G'$ then $N \cap Nam_1 \neq \emptyset$ and $N \cap Nam_2 \neq \emptyset$. Using proof by contradiction, let $(N, g) \notin B \cup D$. Thus, there is an outgoing transition with label (N, g) from q_B in C_1 and an outgoing transition with label (N, g) from q_D in C_2, and based on Definition 3.8, there is an outgoing transition with label (N, g) from q_F in $C_1 \bowtie_C C_2$. But this contradicts that (σ, A) is a failure.

(iii),(iv) These propositions are direct consequences of Definitions 8.5 and 3.8.

(v) By Lemma 8.1(c), $dfail(C_1 \bowtie_C C_2) = sfail(C_1 \bowtie_C C_2) \cup (divtr(C_1 \bowtie_C C_2) \times 2^\Sigma)$. Using 8.4(ii),

$$dfail(C_1 \bowtie_C C_2) = \{(\sigma, A) \mid \exists(\pi, B) \in sfail(C_1),\ \exists(\rho, D) \in sfail(C_2), \\ Join(\sigma;\pi,\rho) \ \wedge \ A \cap G \subseteq B \cap D \ \wedge \ A \cap G' \subseteq B \cup D\} \\ \cup \ (divtr(C_1 \bowtie_C C_2) \times 2^\Sigma). \qquad (**)$$

Equation (**) contains two instances of *sfail* and we need to show that the replacement

of both by $dfail$ do not add any new pair (σ, A) to the righthand side of the equation. In fact, we can show that the replacement of instances of $sfail$ by $dfail$ adds some pairs to the set $\{(\sigma, A)\ |...\}$ in the righthand side of the equation, but all of these new pairs are in $(divtr(C_1 \bowtie_C C_2) \times 2^{\Sigma})$. Thus, the union set (the righthand side of the equation) does not change. For this purpose, first suppose that we replace $sfail(C_1)$ by $dfail(C_1)$. Because, $dfail(C_1) = sfail(C_1) \cup (divtr(C_1) \times 2^{\Sigma})$ (see Lemma 8.1(c)), the only effect of this replacement is that new pairs (σ, A) may be introduced related to some (π, B) and (ρ, D) such that $\pi \in divtr(C_1)$, $(\rho, D) \in sfail(C_2)$ and $Join(\sigma;\pi, \rho)$ holds. But then $\rho \in tr(C_2)$, and by the replacement of $sfail$ by $dfail$, (σ, A) belongs to $(divtr(C_1) \times 2^{\Sigma})$. By a symmetric argument, we can show that the replacement of the other $sfail$ by $dfail$ does not change the righthand side of Equation (**).

(vi) This item is a direct consequence of Definitions 8.5, 3.8 and 8.8. □

Now, we can prove that CFFD is a congruence with respect to join of constraint automata:

Proposition 8.5 Let C and C' be constraint automata over the same set of names and D and D' be constraint automata over the same set of names, such that $C \overset{cffd}{\approx} C'$ and $D \overset{cffd}{\approx} D'$. Then, $C \bowtie_C D \overset{cffd}{\approx} C' \bowtie_C D'$.

Proof. According to Definition 8.6 we need to prove four items:

(i) $stable(C \bowtie_C D) = stable(C) \wedge stable(D)$, based on Proposition 8.4(iii),
$= stable(C') \wedge stable(D')$, because $C \overset{cffd}{\approx} C'$ and $D \overset{cffd}{\approx} D'$,
$= stable(C' \bowtie_C D')$.

(ii) Based on Proposition 8.4(ii),
$sfail(C \bowtie_C D) = \{(\sigma, A) \mid \exists(\pi, B) \in sfail(C), \exists(\rho, E) \in sfail(D), Join(\sigma;\pi, \rho)$
$\wedge A \cap G \subseteq B \cap E \wedge A \cap G' \subseteq B \cup E\}$ where,
$G = \{(N, g) | N \cap Nam_C = \emptyset \vee N \cap Nam_D = \emptyset\}$ and
$G' = \{(N, g) | N \subseteq Nam_C \cup Nam_D \wedge N \neq \emptyset \wedge N \cap Nam_C \neq \emptyset \wedge N \cap Nam_D \neq \emptyset\}$.
Because of the CFFD-equivalence $sfail(C) = sfail(C')$ and $sfail(D) = sfail(D')$. Because of the equality of the names sets, G and G' in the case of $C \bowtie_C D$ are, respectively, equal to G and G' in the case of $C' \bowtie_C D'$, respectively. Thus, $sfail(C \bowtie_C D) = sfail(C' \bowtie_C D')$.

(iii) Based on Proposition 8.4(iv),
$divtr(C \bowtie_C D) = \{\sigma \mid \exists \pi \in tr(C),\ \exists \rho \in tr(D),\ Join(\sigma;\pi, \rho)\ and\ (\pi \in divtr(C) \vee \rho \in divtr(D))\}$. Based on Lemma 8.1(a), $tr(C) = divtr(C) \cup \{\sigma | (\sigma, \emptyset) \in sfail(C)\}$ and this fact holds also for C', D and D'. For CFFD equivalence, it holds that $divtr(C) = divtr(C')$, $divtr(D) = divtr(D')$, $sfail(C) = sfail(C')$, and $sfail(D) = sfail(D')$. Thus, $tr(C) = tr(C')$ and $tr(D) = tr(D')$. Therefore, $divtr(C \bowtie_C D) = divtr(C' \bowtie_C D')$.

(vi) In part (iii) above we proved that $tr(C) = tr(C')$ and $tr(D) = tr(D')$. Also, using the definition of CFFD-equivalence relation, we know that $inftr(C) = inftr(C')$ and

$inftr(D) = inftr(D')$. Thus, using Proposition 8.4(vi), it is the case that $inftr(C \bowtie_C D) = inftr(C' \bowtie_C D')$. □

Thus, CFFD-equivalence is a congruence with respect to the join of constraint automata. A similar result holds also for NDFD-equivalence:

Proposition 8.6 Let C and C' be constraint automata over the same set of names, D and D' be constraint automata over the same set of names, $C \overset{ndfd}{\approx} C'$ and $D \overset{ndfd}{\approx} D'$. Then, $C \bowtie_C D \overset{ndfd}{\approx} C' \bowtie_C D'$.

Proof.
The proofs for the claims $stable(C \bowtie_C D) = stable(C' \bowtie_C D')$, $divtr(C \bowtie_C D) = divtr(C' \bowtie_C D')$ and $inftr(C \bowtie_C D) = inftr(C' \bowtie_C D')$ are similar to the proofs of their counterparts in Proposition 8.5. (We use $dfail$ sets instead of $sfail$ sets and part (b) of Lemma 8.1 instead of part (a) to show the trace equivalences.) Now we prove that, $dfail(C \bowtie_C D) = dfail(C' \bowtie_C D')$. By Proposition 8.4(v),
$dfail(C \bowtie_C D) = \{(\sigma, A) \mid \exists(\pi, B) \in dfail(C),\ \exists(\rho, E) \in dfail(D),\ Join(\sigma;\pi, \rho)$
and $A \cap G \subseteq B \cap E \ \wedge\ A \cap G' \subseteq B \cup E\} \ \cup\ (divtr(C_1 \bowtie_C C_2) \times 2^{\Sigma})$.
Because $C \overset{ndfd}{\approx} C'$ and $D \overset{ndfd}{\approx} D'$, $dfail(C) = dfail(C')$, $dfail(D) = dfail(D')$ and $divtr(C \bowtie_C D) = divtr(C' \bowtie_C D')$. Because of the equality of the names sets, G and G' in the case of $C \bowtie_C D$ are, respectively, equal to G and G' in the case of $C' \bowtie_C D'$. Thus, $dfail(C \bowtie_C D) = dfail(C' \bowtie_C D')$. □

Therefore, NDFD-equivalence is a congruence with respect to the join of constraint automata.

8.4 Congruency Results for Hiding Names

In this section we prove that the equivalence relation CFFD is a congruence with respect to hiding of port names in constraint automata (with τ-transitions) and that is also the case for the equivalence relation NDFD. Our method of proof is a modification and extension of the proof of that CFFD and NDFD relations are congruences for the case of hiding of an alphabet member in all transitions of an LTS presented in [142].

First, we show that the sets of finite or infinite traces, stable failures, divergent traces and divergence-masked failures of the automaton after hiding of a port name can be characterized by their counterparts in the original constraint automaton. Based on these characterizations, we prove our congruency results.

Definition 8.9

Let Nam be a set of names, $Data$ be a set of data, $\Sigma = \{(N, g) | N \subseteq Nam \ \wedge\ g \in DC(N, Data)\}$ and $B \in Nam$. We define the set **hide** B **in** Σ_1, for every set $\Sigma_1 \subseteq \Sigma$ such that:

$$\textbf{hide } B \textbf{ in } \Sigma_1 \ = \ \{(N \backslash \{B\}, \exists B[g]) |\ (N, g) \in \Sigma_1\} \setminus \{\tau\},$$

where for data constraint g, we define $\exists B[g] = \vee_{d \in \mathcal{D}} g[d_B/d]$ (see Definition 3.9).

Also, for every finite or infinite string $\sigma = (N_1, g_1)(N_2, g_2)\ldots$ we define the string **hide** B **in** σ as the string that is obtained by removing all pairs of the form $(\emptyset, g)$ from the word $(N_1 \backslash \{B\}, \exists B[g_1])(N_2 \backslash \{B\}, \exists B[g_2])\ldots$.

The following proposition lists some basic results which we need in the proof of other theorems:

Proposition 8.7
Let $C = \langle Q, Nam, T, q_0 \rangle$ be a constraint automaton, $B \in Nam$ be a port name, and $\exists B[C]$ be the constraint automaton resulting from hiding of B in C (see Definition 3.10). Then,
(i) $tr(\exists B[C]) = \{$**hide** B **in** $\sigma |\ \sigma \in tr(C)\}$.
(ii) $sfail(\exists B[C]) = \{(\textbf{hide}\ B\ \textbf{in}\ \sigma, A) |\ (\sigma, A \cup A' \cup \widehat{B}) \in sfail(C)\}\}$, where
$A' = \{(N \cup \{B\}, g) |\ \exists g' \in DC(N, data){:}(N, g') \in A\}$,
$\widehat{B} = \{(\{B\}, g) |\ g \in DC(\{B\}, data)\}$.
(iii) $stable(\exists B[C]) = stable(C) \wedge \forall g \in DC(\{B\}, Data){:}(\{B\}, g) \notin tr(C)$.
(iv) $divtr(\exists B[C]) = \{$**hide** B **in** $\sigma |\ \sigma \in divtr(C)\} \cup$
$\{$**hide** B **in** $\sigma | \sigma \in inftr(C)\ \wedge |$**hide** B **in** $\sigma| < \infty\}$.
(v) $dfail(\exists B[C]) = \{(\textbf{hide}\ B\ \textbf{in}\ \sigma, A) |\ (\sigma, A \cup A' \cup \widehat{B}) \in dfail(C)\}\ \cup$
$(divtr(\exists B[C]) \times 2^{\Sigma})$, where, Σ is so defined in Definition 8.9.
(vi) $inftr(\exists B[C]) = \{$**hide** B **in** $\omega |\ \omega \in inftr(C) \wedge |$**hide** B **in** $\omega| = \infty\}$.

Proof.
(i) This is a direct consequence of Definitions 8.5 and 8.9.

(ii) If $(\rho, A) \in sfail(\exists B[C])$, then for the automaton $(\exists B[C])$, we know that there is a state $q \in Q$ where $q_{0,B} \stackrel{\rho}{\Longrightarrow} q$ and $stable(q)$ and $\forall a \in A\ (\neg\ q \stackrel{a}{\Longrightarrow})$. Because ρ is a trace in $\exists B[C]$, there is a trace $\sigma \in tr(C)$ such that $\rho =$ **hide** B **in** σ, $\Sigma(\rho) =$ **hide** B **in** $\Sigma(\sigma)$ and in the automaton C, $q_0 \stackrel{\sigma}{\Longrightarrow} q$. Because, q is stable in $\exists B[C]$, there is no transition of the form $q \stackrel{\tau}{\rightarrow}_B q'$, and using the definition of hiding, there is no transition of the form $q \stackrel{\tau}{\rightarrow} q'$ in C. Thus, q is also stable in C. Now we prove that $(\sigma, A \cup A' \cup \widehat{B})$ is a failure of C. First, note that because (ρ, A) is a failure of $\exists B[C]$, for all $(N, g) \in A$, $B \notin A$. Thus A and A' are two disjoint sets. Because (ρ, A) is a failure in $\exists B[C]$ and $\rho =$ **hide** B **in** σ, (σ, A) is a failure of C. For the set A', we know that $A =$ **hide** B **in** A'. Thus, (σ, A') is also a failure of C. Because q is stable in $\exists B[C]$, by the definition of hiding, there is no transition of the form $q \stackrel{\{B\},g}{\longrightarrow} q'$ in C. Thus, $(\sigma, \widehat{B})$ is a failure of C. It follows that, $sfail(\exists B[C]) \subseteq \{(\textbf{hide}\ B\ \textbf{in}\ \sigma, A) |\ (\sigma, A \cup A' \cup \widehat{B})\}$.
On the other hand, let $(\sigma, A \cup A' \cup \widehat{B}) \in sfail(C)\}$ and $\rho =$ **hide** B **in** σ. Thus, for the automaton C, we know that there is a state $q \in Q$ where $q_0 \stackrel{\sigma}{\Longrightarrow} q$ and $stable(q)$ and $\forall a \in A \cup A' \cup \widehat{B},\ \ (\neg\ q \stackrel{a}{\Longrightarrow})$. Because $q_0 \stackrel{\sigma}{\Longrightarrow} q$ is a run of C and $\rho =$ **hide** B **in** σ, $q_{0,B} \stackrel{\rho}{\Longrightarrow} q$ is a run of $\exists B[C]$. Because in the automaton C there is no transition of the form $q \stackrel{a}{\rightarrow} q'$ in which $a \in A \cup A'$, by using the definition of hiding, there is no transition of the form $q \stackrel{a}{\rightarrow} q'$ in which, $a \in A$ in the automaton $\exists B[C]$. Thus, (ρ, A) is a failure of $\exists B[C]$. Because q is stable in C and there is no transition of the form $q \stackrel{a}{\rightarrow} q'$,

$a \in \{(\{B\}, g) | g \in DC(\{B\}, data)\}$, and q is stable in $\exists B[C]$. Thus, (ρ, A) is a stable failure of $\exists B[C]$. Therefore, $\{(\textbf{hide } B \textbf{ in } \sigma, A) | \ (\sigma, A \cup A' \cup \widehat{B})\} \subseteq \ sfail(\exists B[C])$.

(iii),(iv) These are direct consequences of Definitions 8.5 and 3.10.

(v) By Lemma 8.1(c), $dfail(\exists B[C]) = sfail(\exists B[C]) \cup (divtr(\exists B[C]) \times 2^{\Sigma})$. Thus, using 8.7(ii),

$dfail(\exists B[C]) =$
$\{(\textbf{hide } B \textbf{ in } \sigma, A) | \ (\sigma, A \cup A' \cup \widehat{B}) \in sfail(C)\} \cup (divtr(\exists B[C]) \times 2^{\Sigma}) \qquad (*)$

The effect of the replacement of $sfail$ by $dfail$ in Equation (*) is that new pair $(\textbf{hide } B \textbf{ in } \sigma, A)$ may be introduced where $\sigma \in divtr(C)$. But by Definition 8.9, if $\sigma \in divtr(C)$ then $\textbf{hide } B \textbf{ in } \sigma \in divtr(\exists B[C]))$. Thus, the replacement of $sfail$ by $dfail$ in Equation (*) does not change its righthand side.
(vi) This is a direct consequence of Definitions 8.5 and 8.9. □

Now, we can prove that CFFD is a congruence with respect to hiding of port names of constraint automata:

Proposition 8.8 Let C and C' be constraint automata over the same set of names, $C \overset{cffd}{\approx} C'$ and B be a name in the set of names. Then, $\exists B[C] \overset{cffd}{\approx} \exists B[C']$.

Proof. (i) By Proposition 8.7(iii),
$stable(\exists B[C]) = stable(C) \wedge \forall g \in DC(\{B\}, Data)\text{:}(\{B\}, g) \notin tr(C)$.
Because $C \overset{cffd}{\approx} C'$, $stable(C) = stable(C')$, $divtr(C) = divtr(C')$ and $sfail(C) = sfail(C')$. By Lemma 8.1(a), $tr(C) = divtr(C) \cup \{(\sigma, \emptyset) | \sigma \in sfail(C)\}$. Thus, $tr(C) = tr(C')$. Therefore, $stable(\exists B[C]) = stable(\exists B[C'])$.

(ii) By Proposition 8.7(ii),

- $sfail(\exists B[C]) = \{(\textbf{hide } B \textbf{ in } \sigma, A) | \ (\sigma, A \cup A' \cup \widehat{B}) \in sfail(C)\}\}$,
- $A' = \{(N \cup \{B\}, g) | \ \exists g' \in DC(N, data)\text{:}(N, g') \in A\}$,
- $\widehat{B} = \{(\{B\}, g) | \ g \in DC(\{B\}, data)\}$.

Because $C \overset{cffd}{\approx} C'$, $sfail(C) = sfail(C')$. Because the name sets of C and C' are equal, the definitions of sets A' and $\widehat{B}$ in the cases of C and C' are the same. Thus, $sfail(\exists B[C]) = sfail(\exists B[C'])$.

(iii) By Proposition 8.7(iv), $divtr(\exists B[C])$ is equal to $\{\textbf{hide } B \textbf{ in } \sigma | \ \sigma \in divtr(C)\} \cup \{\textbf{hide } B \textbf{ in } \sigma | \sigma \in inftr(C) \wedge |\textbf{hide } B \textbf{ in } \sigma| \langle \infty\}$. Because $C \overset{cffd}{\approx} C'$, $inftr(C) = inftr(C')$ and $divtr(C) = divtr(C')$. Therefore, $divtr(\exists B[C]) = divtr(\exists B[C'])$.

(iv) Because $C \overset{cffd}{\approx} C'$, $inftr(C) = inftr(C')$. Thus, using Proposition 8.7(vi), we know that $inftr(\exists B[C]) = inftr(\exists B[C'])$. □

Therefore, CFFD-equivalence is a congruence with respect to the hiding of port names in constraint automata. Similarly, we prove that NDFD is a congruence with respect to the hiding operator for constraint automata:

Proposition 8.9 Let C and C' be constraint automata over the same set of names, $C \overset{ndfd}{\approx} C'$ and B be a name in the set of names. Then, $\exists B[C] \overset{ndfd}{\approx} \exists B[C']$.

Proof. The proofs for claims:
$stable(\exists B[C]) = stable(\exists B[C'])$, $divtr(\exists B[C]) = divtr(\exists B[C'])$ and
$inftr(\exists B[C]) = inftr(\exists B[C'])$ are similar to the proofs of their counterparts in Proposition 8.8. Further, by Proposition 8.7(v),
$dfail(\exists B[C]) = \{(\textbf{hide } B \textbf{ in } \sigma, A) | \ (\sigma, A \cup A' \cup \widehat{B}) \in dfail(C)\} \cup (divtr(\exists B[C]) \times 2^{\Sigma})$.
Because $C \overset{ndfd}{\approx} C'$, $dfail(C) = dfail(C')$. As we showed, $divtr(\exists B[C]) = divtr(\exists B[C'])$.
Thus, $dfail(\exists B[C]) = dfail(\exists B[C'])$. □

Thus, NDFD-equivalence is a congruence with respect to the hiding of port names in constraint automata.

8.5 Linear Temporal Logic of Constraint Automata

Traditionally temporal logics are logical systems for specification and verification of the properties that are based on the truth values of propositions in the states of a transition system. Such transition systems are called Kripke structures. Linear models (see Definition 8.10) are simplifications or runs of Kripke structures. On the other hand, labeled transition systems and constraint automata are transition systems with labels on their transitions. Also, process algebraic equivalences and composition operators usually work purely on information that is based on transition labels. In this section, we augment the definitions of labeled transition systems and constrain automata by introducing functions that assign to each of their states a set of propositions. Then, we introduce linear temporal logic and two of its fragments interpreted over linear models as executions of augmented labeled transition systems or augmented constraint automata.

Definition 8.10
(i) Let AP be a set of atomic propositions. A *Linear Model* is a finite or infinite sequence $\sigma = \sigma_1, \sigma_2, \ldots$ of subsets of AP. We call any $\sigma_i \subseteq AP$ a state of (in) the linear model σ.
(ii) An *augmented labeled transition system (aLTS)* is a 5-tuple $A = \langle S, s, \Delta, AP, L \rangle$, where, $\langle S, s, \Delta \rangle$ is an LTS, AP is a set of propositions, and $L{:}S \rightarrow 2^{AP}$ is a labeling function. Let $\sigma \in \Sigma^{\omega}$ be an infinite trace of the LTS $\langle S, s, \Delta \rangle$. Because σ is an infinite trace, there is an infinite (or deadlocking) sequence of LTS (S, s, Δ), of the form $r = (s, \sigma_1, s_1), (s_1, \sigma_2, s_2), \ldots$. The *linear model* defined by r in A is $M_r = L(s), L(s_1), L(s_2), \ldots$.

(iii) A tuple $C = \langle Q, Nam, T, q_0, AP, L\rangle$ is called as an *augmented constraint automaton* (aCA) where $\langle Q, Nam, T, q_0\rangle$ is a constraint automaton, AP is a set of propositions, and $L{:}Q \rightarrow 2^{AP}$ is a labeling function. Let C be an aCA and $r = (q_0, \phi_1, q_1), (q_1, \phi_2, q_2), \ldots$ be an infinite or deadlocking run of C. The *linear model* defined by r in C is $M_r = L(q_0), L(q_1), L(q_2), \ldots$.

Now, we present the syntax and semantics of linear temporal logic and two of its fragments:

Syntax of LTL and its fragments

(i) The set of all well-formed formulas of *linear temporal logic (LTL)* is defined by the following abstract syntax:

$$\phi ::= \ P \mid \neg\phi \mid \phi \vee \phi \mid \phi U \phi \mid X\phi \qquad P \in AP$$

(ii) The set of all well-formed formulas of *Next-time-less linear temporal logic (LTL_{-X})* is defined by the following abstract syntax:

$$\phi ::= \ P \mid \neg\phi \mid \phi \vee \phi \mid \phi U \phi \qquad P \in AP$$

(iii) The set of all well-formed formulas of *restricted linear temporal logic (LTL_{ω})* is defined by the following abstract syntax:

$$\phi ::= \ P \mid \neg\phi \mid \phi \vee \phi \mid \phi U \phi \mid \overset{\omega}{F}\ \phi \qquad P \in AP$$

We also use the following abbreviations:

$$\top \equiv_{df} (p \vee (\neg p))$$

where p is a fixed proposition,

$$\phi_1 \wedge \phi_2 \equiv_{df} \neg(\neg\phi_1 \vee \neg\phi_2),$$

$$F\phi \equiv_{df} \top U \phi,$$

and

$$G\phi \equiv_{df} \neg F(\neg\phi).$$

Semantics of LTL and its fragments

A temporal formula ϕ of the above defined syntactic structures holds in a linear model σ (denoted by $\sigma \vDash \phi$) according to the following rules:

1- If $\phi \in AP$, then $\sigma \vDash \phi$ iff $\phi \in \sigma_1$.
2- $\sigma \vDash \neg\phi$ iff not $\sigma \vDash \phi$.
3- $\sigma \vDash (\phi_1 \vee \phi_2)$ iff $\sigma \vDash \phi_1$ or $\sigma \vDash \phi_2$
4- $\sigma \vDash (\phi_1 U \phi_2)$ iff $\exists i{:}0 \leq i < |\sigma|, \sigma^i \vDash \phi_2$ *and* $\forall j{:}0 \leq j < i, \sigma^j \vDash \phi_1$.
5- $\sigma \vDash X\phi$ iff $\sigma^1 \neq \emptyset$ and $\sigma^1 \vDash \phi$.
6- $\sigma \vDash \overset{\omega}{F}\ \phi$ iff there are infinitely many $i \geq 0$ such that $\sigma^i \vDash \phi$.

In terms of expressiveness power, it can be shown that $LTL_{-X} \subset LTL_{\omega} \subset LTL$. In general, in LTL we have, $\overset{\omega}{F}\ \phi \equiv GXF\phi$, but if we restrict to infinite linear models only then $\overset{\omega}{F}\ \phi \equiv GF\phi$. Therefore, the temporal operator $\overset{\omega}{F}$ is an operator for distinguishing a finite linear model from an infinite one, i.e., distinguishing a *deadlock* from a *divergence*.

Definition 8.11 Let $\sigma = \sigma_1, \sigma_2, \ldots$ be a linear model.
(i) The *finitely reduced form* of σ (denoted by $fred(\sigma)$) is constructed by collapsing all finite continuous sequences $\sigma_i, \sigma_{i+1}, \ldots, \sigma_{i+m}$ of identical elements $\sigma_i = \sigma_{i+1} = \ldots = \sigma_{i+m}$ to one element σ_i.
(ii) The *reduced form* of σ (denoted by $red(\sigma)$) is constructed by collapsing all finite and infinite continuous sequences $\sigma_i, \sigma_{i+1}, \ldots$ of identical elements $\sigma_i = \sigma_{i+1} = \ldots$ to one element σ_i.
(iii) If σ_1 and σ_2 are two linear models, we say that σ_1 and σ_2 are *equivalent under stuttering* iff $red(\sigma_1) = red(\sigma_2)$.

By induction on the syntactic structure of formulas, we obtain the following proposition.

Proposition 8.10 Let $\sigma = \sigma_1, \sigma_2, \ldots$ be a linear model.
(i) If ϕ is an LTL_{ω}-formula, then $\sigma \vDash \phi$ iff $fred(\sigma) \vDash \phi$.
(ii) If ϕ is an LTL_{-X}-formula, then $\sigma \vDash \phi$ iff $red(\sigma) \vDash \phi$.

In the context of model checking, we use aLTSs and aCAs as the models of our systems and also as the semantic domain of our temporal logic. On the other hand, we want to use of the equivalence relations to reduce the models' sizes. This equivalence based reduction will be useful in model checking if the reduction process preserves the truth values of each temporal logic formula. Now we intend to formally define the concept of preservation of the truth values of temporal formulas according to each equivalence relation. For this, we can use a way of interpreting the transition labels as functional state transformers [86]. In this section, we use this transformation only for defining the concept of truth preservation, but in the next section we will use a modified version of it in our reduction algorithm.

Definition 8.12
(i) A *state modifier* sm is a mapping $sm{:}2^{AP} \rightarrow 2^{AP}$. The set of all state modifiers is denoted by TS. The identity state modifier I is the identity function. A *state modifier sequence* is a finite or infinite sequence of state modifiers.
(ii) A *temporal semantics* for an LTS or constraint automaton L is a mapping $f{:}\Sigma(L) \cup \{\tau\} \rightarrow TS$ such that $f(\tau) = I$. If $\rho = a_1 a_2 \ldots$ is a path of L, we write $f(\rho)$ for the sequence $(f(a_1), f(a_2), \ldots)$. A temporal semantics for a path ρ is a mapping $f{:}\Sigma(\rho) \cup \{\tau\} \rightarrow TS$ such that $f(\tau) = I$.
(iii) The linear model induced by a state $\nu \subseteq AP$ and a state modifier sequence sms, denoted as $Model(\nu, sms)$, is a sequence of states such that:
1- $Model(\nu, sms)_0 = \nu$
2- $Model(\nu, sms)_{i+1} = sms_i(Model(\nu, sms)_i)$.
If sms is finite then $|Model(v, sms)| = |sms| + 1$.
(ix) Let $\sigma \in (\Sigma_{\tau}^{*} \cup \Sigma_{\tau}^{\omega})$ be a path of an LTS L, f a temporal semantics for σ, ν_0 a state, and ϕ an LTL formula. We say ϕ is true of σ with respect to temporal semantics f and initial state ν_0 and write $\sigma, f, \nu_0 \vDash \phi$ iff $Model(\nu_0, f(\sigma)) \vDash \phi$.

Usually, linear temporal logic formulas are interpreted over the complete paths generated by a transition system. These correspond to the infinite and deadlocking paths of an LTS.

Definition 8.13 (i) Let L be an LTS, f a temporal semantics for L, ν_0 a state, and ϕ an LTL formula. We say ϕ is true of L with respect to temporal semantics f and initial state ν_0, and write $L, f, \nu_0 \models \phi$ iff $\sigma, f, \nu_0 \models \phi$ for all $\sigma \in dpath(L) \cup infpath(L)$.
(ii) Let L_1 and L_2 be LTSs and ϕ an LTL-formula. We say that L_1 and L_2 agree on ϕ iff for every temporal semantics f and for every initial state ν_0 it is the case that $L_1, f, \nu_0 \models \phi$ iff $L_2, f, \nu_0 \models \phi$.
(iii) An equivalence $\approx$ between LTSs is *LTL-preserving* iff for any pair L_1 , L_2 such that $L_1 \approx L_2$, L_1 and L_2 agree on every LTL formula. Similarly, An equivalence $\approx$ between LTSs is LTL_{-X} (LTL_ω)-preserving iff for any L_1 , L_2 such that $L_1 \approx L_2$, L_1 and L_2 agree on every LTL_{-X} (LTL_ω) formula.

Let L be a labeled transition system. Intuitively, a temporal semantics for L expresses the changes caused by the transitions in the information contained in each state of L. But, L can be composed with other labeled transition systems using composition operators defined in Definitions 8.2 and 8.3, and in addition, in the case of constraint automata, using join and hiding operators defined in Chapter 3. Thus, we need to define how a composition operator affects the temporal semantics of the original labeled transition systems, which will be composed using these operators.

For the composition operators defined in in Definitions 8.2 and 8.3, all temporal semantics for compositional labeled transition systems have been defined in [86]. Also, it was shown in [86] that:

Proposition 8.11 For each labeled transition system and with respect to all composition operators that have well defined temporal semantics:
(i) CFFD-equivalence is LTL_ω-preserving and NDFD-equivalence is LTL_{-X}-preserving.
(ii) If $\approx$ is an equivalence between LTSs and it is congruence with respect to $||[\cdots]||$ and $[]$ (defined in Definition 8.2) and is LTL_ω-preserving, then $L \approx L'$ implies $L \overset{cffd}{\approx} L'$. Thus, CFFD is the weakest compositional equivalence preserving LTL_ω.
(iii) If $\approx$ is an equivalence between LTSs and it is congruence with respect to $||[\cdots]||$ and $[]$ and is LTL_{-X}-preserving, then $L \approx L'$ implies $L \overset{ndfd}{\approx} L'$. Thus, NDFD is the weakest compositional equivalence preserving LTL_{-X}.

The proof of Proposition 8.11(i) depends only on the definitions of the equivalences, temporal semantics, and the notion of temporal logic preservation (see [86]). According to Proposition 8.11(ii),(iii), the minimality property holds whenever an arbitrary equivalence $\approx$ is a congruence with respect to the parallel composition $||[\cdots]||$ and non-deterministic choice $[]$ operators. We have shown that every constraint automaton $C = \langle Q, Nam, T, q_0 \rangle$ can be considered as a labeled transition system with alphabet $\Sigma = \{(N, g) | N \subseteq Nam \wedge g \in DC(N, Data) \wedge N \neq \emptyset\}$ (Proposition 8.3) and proved that CFFD and NDFD-equivalences are congruences with respect to our defined join and hiding operators for constraint automata. Thus, if we can define the temporal semantics of the composed constraint automaton by means of the temporal semantics of the original automata, then all parts of Proposition 8.11

will hold not only for constraint automata composed by operators defined in Definitions 8.2 and 8.3, but also when they are composed by join and hiding defined in Chapter 3.

Now we investigate the effects of join and hiding composition operators for constraint automata on their temporal semantics. First, we need to make precise the meaning of *effects* of a composition operator on a temporal semantics:

Definition 8.14
(i) A state modifier sm affects an atomic proposition a iff there is a $\nu \subseteq AP$ such that either $a \in \nu$ and $a \notin sm(\nu)$, or $a \notin \nu$ and $a \in sm(\nu)$. We denote by $af(sm)$ and $\overline{af}(sm)$, the set of all atomic propositions affected by sm and the set $AP \backslash af(sm)$.
(ii) State modifiers sm and sm' are *compatible* iff for all atomic proposition $a \in af(sm) \cap af(sm')$ and all $\nu \subseteq AP$, $a \in sm(\nu)$ iff $a \in sm'(\nu)$. If this is the case, the *combination* of sm and sm', denoted by $sm \oplus sm'$, is the function $c{:}2^{AP} \rightarrow 2^{AP}$ where $c(\nu) = (sm(\nu) \cap af(sm)) \cup (sm'(\nu) \cap af(sm')) \cup (\nu \cap \overline{af}(sm) \cap \overline{af}(sm'))$.

Definition 8.15 Let $C_1 = \langle Q_1, Nam_1, T_1, q_{01} \rangle$ and $C_2 = \langle Q_2, Nam_2, T_2, q_{02} \rangle$ be two constraint automata, and f_1 and f_2 be temporal semantics for C_1 and C_2, respectively. f_1 and f_2 are *compatible with respect to synchronization set* $G = \{(N, g) | \ (N = N_1 \cup N_2) \wedge (g = g_1 \wedge g_2) \wedge (N_1 \neq \emptyset) \wedge (N_2 \neq \emptyset) \wedge (N_1 \subseteq Nam_1) \wedge (N_2 \subseteq Nam_2) \wedge (g_1 \in DC(Nam_1, Data)) \wedge (g_2 \in DC(Nam_2, Data)) \wedge (N_1 \cap Nam_2 = N_2 \cap Nam_1)\}$, iff for all $g \in G$, $f_1(g)$ and $f_2(g)$ are compatible and for all $l \in (\Sigma(C_1) \cap \Sigma(C_2)) \backslash G$, $f_1(l) = f_2(l)$.

Let $C = C_1 \bowtie_C C_2$ and f_1 and f_2 be temporal semantics for C_1 and C_2, respectively. The state information of C consists of the state information of both C_1 and C_2. The temporal semantics f_1 expresses changes in the state information of C_1 and f_2 that of C_2. These changes are made by transitions. Thus, if a transition in C corresponds to a transition of C_1 alone, the change in the state information of C is the same as in C_1. Also, if a transition in C corresponds to a transition of C_2 alone, the change in the state information in C is the same as in C_2. But, if a transition in C corresponds to synchronized transitions of C_1 and C_2, the change in the state information of C should consist of both changes in C_1 and C_2. The set G defined in Definition 8.15 identifies the set of all synchronization alphabets. If the temporal semantics f_1 and f_2 are mutually conflicting, it is impossible to define a joint temporal semantics in the case of synchronization. In Definition 8.15, the compatibility requirement guarantees the possibility of joining two temporal semantics. Thus, based on the above definition, the temporal semantics for $C_1 \bowtie_C C_2$ can be characterized as:

Proposition 8.12 Let $C_1 = \langle Q_1, Nam_1, T_1, q_{01} \rangle$ and $C_2 = \langle Q_2, Nam_2, T_2, q_{02} \rangle$ be two constraint automata, f_1 and f_2 be temporal semantics for C_1 and C_2, respectively, and let f_1 and f_2 are compatible with respect to set G (defined in Definition 8.15). The temporal semantics for $C = C_1 \bowtie_C C_2$ is the function f such that:
$\forall l \in G{:}f(g) = f_1(l) \oplus f_2(l)$, $\forall l \in \Sigma(C_1) \backslash G{:}f(l) = f_1(l)$ and $\forall l \in \Sigma(C_2) \backslash G{:}f(l) = f_2(l)$.

Let $\exists B[C]$ be the constraint automaton resulting from hiding of $B \in Nam$ in constraint automaton $C = (Q, Nam, T, q_0)$, and f_1 be a temporal semantics for C. To characterize the temporal semantics of $\exists B[C]$, first note that every transition label of the form $(\{B\}, g)$ in C is a transition with the label τ in $\exists B[C]$. Because τ-transitions do not affect information of states (see Definition 8.12(ii)), it must be that $f_1(l) = I$, for all $l \in \{(\{B\}, g) | g \in$

$DC(Nam, Data)\}$, where I is the identity function. If this condition holds, the temporal semantics of $\exists B[C]$ must do the same changes to the information of states as the temporal semantics of C does. This can be expressed by: $\forall(N, g) \in \Sigma(C)$:$f_1((N \backslash \{B\}, \exists B[g])) = f((N, g))$ and $f_1(\tau) = I$. Thus:

Proposition 8.13 Let $C = \langle Q, Nam, T, q_0 \rangle$ be a constraint automaton, $B \in Nam$, and f_1 be a temporal semantics for C. The temporal semantics for $\exists B[C]$ can be defined iff $\forall l \in \{(\{B\}, g) | g \in DC(Nam, Data)\}$:$f_1(l) = I$. If this is the case, then, $\forall(N, g) \in \Sigma(C)$:$f_1((N \backslash \{B\}, \exists B[g])) = f((N, g))$ and $f_1(\tau) = I$.

Based on Propositions 8.12 and 8.13, we have a well-defined temporal semantics for the join and hiding operators of constraint automata. Because of our translation of constraint automata to labeled transition systems, we have the followings:

Proposition 8.14 For each constraint automaton and with respect to all composition operators defined in Definitions 8.2 and 8.3 extended with the join and hiding operators defined in Chapter 3:
(i) CFFD-equivalence is LTL_ω-preserving and NDFD-equivalence is LTL_{-X}-preserving.
(ii) If $\approx$ is an equivalence between constraint automata and it is congruence with respect to $||[\cdots]||$ and $[]$, and it is LTL_ω-preserving, then $C \approx C'$ implies $C \overset{cffd}{\approx} C'$. Thus, CFFD is the weakest compositional equivalence over the set of constraint automata preserving LTL_ω.
(iii) If $\approx$ is an equivalence between constraint automata and it is congruence with respect to $||[\cdots]||$ and $[]$, and it is LTL_{-X}-preserving, then $C \approx C'$ implies $C \overset{ndfd}{\approx} C'$. Thus, NDFD is the weakest compositional equivalence over the set of constraint automata preserving LTL_{-X}.

8.6 Reduction Algorithms

The process of model checking contains three main steps: 1- Modeling of the actual system using a formal system such as aLTS or aCA. 2- Expressing the requirement or property that we want to verify by using a formula of a temporal logic. 3- Using a model checking algorithm for deciding if the formula is true in the model or not. In our method for model checking of an aLTS or aCA, before doing the third step, we need to reduce the size of the model by using an equivalence relation. This reduction process can be done before or after defining the property or formula that we need to verify. Thus, the reduction can be done before the second or the third step of the model checking process. In this section we present some algorithms for reducing the sizes of aLTSs and aCAs, while preserving NDFD and CFFD-equivalences. The method that we present here is a modification of the algorithm introduced in [141, 86]. Then, we consider the case where we reduce the model after defining a property or a set of properties that we need to verify.

The algorithms for minimizing an aLTS $A = \langle S, s, \Delta, AP, L \rangle$ (or an aCA) with respect to CFFD and NDFD-equivalences have three main steps:

1- Converting the aLTS or aCA A into an *acceptance graph* AG, which relies on the process of converting a finite automaton to its deterministic counterpart. Each node of the graph AG contains a set of states $D \subseteq S$. For each node of the graph AG all states in D are reachable from an initial state by using the same finite divergence trace.

2- Labeling of the nodes of the acceptance graph (deterministic automaton) with the information about stability, divergences, stable failures and non-divergent failures (see [141] for the detail of this part of the labeling process). We also label each node of the acceptance graph with a set of propositions that can be true in it. To determine this set of propositions, let n be a node of the acceptance graph AG that contains the set of states $D \subseteq S$. Let P be the union set of all $L(d)$ where $d \in D$. Obviously, for every $p \in P$, there is a finite trace in aLTS A in the last state of which the proposition p is true. Thus, we label the node n with the set P.

3- Minimizing the acceptance graph (labeled deterministic automaton) by using traditional algorithms for minimizing finite automata. In this step, we must partition the set of all states (nodes). This first level partitioning is done by considering both the propositions that hold in states and the requirements of the intended equivalence. Two states are in the same class, if the sets of propositions that hold in them are compatible. Also, in the case of the CFFD-equivalence, two states with the same stability, divergent traces, and stable failures belong to the same class. In the case of the NDFD-equivalence, two states with the same stability, divergent traces, and divergence-masked failures belong to the same class.

In practice, the main advantage of reducing models with respect to an equivalence, without considering any property to be verified, is that we can run the reduction algorithm once and use the minimized models whenever we need to model check a property. The property must be expressible in a temporal logic that the equivalence preserves it. But suppose that we need to model check a formula or a set of formulas whose set of atomic propositional constituents is $A \subset AP$ and $|A| \ll |AP|$ (the size of A is very much smaller than the size of AP). In cases like this, the above method is not efficient in practice, because in the first phase of partitioning (in step 3), several states that agree on the truth values of the members of A may be allocated in different classes based on their different truth values of the other members of AP. This implies that the size of the model is not reduced much. Thus, in such cases, we first define the property or the set of properties that we need to verify, and then filter all sets of propositions assigned to the states of the model such that they contain only subsets of A. We call the resulting model a *filtered model*. Then, we run the above reduction method on the filtered model.

From the worst case complexity analysis point of view, it can be shown that all of the above reduction algorithms are exponential in the size of the input model. This is true not only for reductions based on CFFD and NDFD-equivalences, but also for a wide range of equivalences and simulation relations defined in automata theory, graph theory, Petri Nets, and process algebras [35]. Experience in all of these fields indicates that, in practice, the worst case rarely happens (for more references to these experiences and a detailed discussion about the complexity of failure based equivalences see [140]).

8.7 Compositional Model Checking

In this section we present a method for compositional model checking of a component-based system and its coordinating subsystem using the above mentioned equivalences to minimize their formal models. A component-based system has two main parts: a set of components and a coordinating subsystem (glue code). Using Reo specifications, one can specify or model the coordinating subsystems in a compositional and hierarchal way. Using constraint automata, not only the coordinating subsystem, but also all components can be modeled as constraint automata in a compositional way. Thus, the methods of compositional reasoning not only can be applied on the coordinating subsystem , but also on the whole component-based system. Fortunately, our above process algebraic discussions enable us to use the equivalence based compositional reduction method in both cases.

Verification of coordinating subsystem. In this case we need to verify the desired properties of the coordinating subsystem of a component-based system. If we consider the coordinating subsystem (for example a Reo circuit or a constraint automaton) as a complete system, the set of the components of the component-based system is its environment. Externally visible actions of this coordinating subsystem are the *read (input or get)* and *write (output or put)* operations it uses to communicate with the environment. (In Reo these operations involve only the boundary nodes of the circuite.) The rest of the actions within the coordinating subsystem, and its internal states are not interesting, if only the correct functionality of the coordinating subsystem is of concern. Thus, the main steps of model checking of the desired properties of the coordinating subsystem consist of:

1- Modeling the behavior of connectors and the observable behavior of components by augmented constraint automata. Because in this case all actions are considered as visible, none of the constraint automata models have τ-transitions.

2- Expressing the desired property by an LTL_{-X} or LTL_{ω} formula.

3- According to the type of the property to verify, using an equivalence relation for reducing the constraint automata models.

4- Composing the reduced constraint automata models using join and hiding operators. Because we proved that CFFD and NDFD are congruences for all composition operators defined in this paper, the composed model will be reduced by itself.

5- Use one of the ordinary LTL model checking algorithms on the minimized model (for the algorithms of LTL model checking see [50]).

Note that because of the minimization, the efficiency of our method is better than applying LTL model checking algorithms directly. Moreover, according to step 4 above, any improvement in the ordinary LTL model checking algorithms, improves the efficiency of our method.

Example 8.1 (Dining Philosophers) The classical dining philosophers problem can be described as a coordination system in Reo [14]. This system can be designed as a set of pairs of instances of two components: philosopher and chopstick. As illustrated in Figures 8.1(a) and (b), the interface of philosopher i has four output ports: lt_i, rt_i, lf_i and rf_i, which serve to take and return the chopsticks on the left- and right-hand sides of the philosopher.

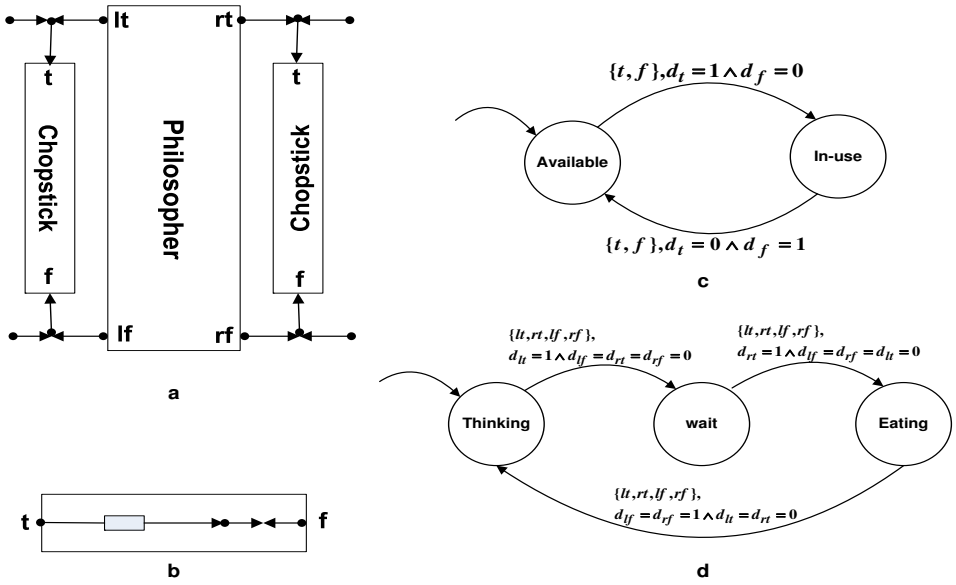


Figure 8.1: (a) Dining philosophers scenario in Reo and (b) a chopstick, (c) minimized constraint automaton for a chopstick and (d) a philosopher

The externally observable behavior of a philosopher component is as follows. After some period of thinking, it decides to eat, attempts to obtain its two chopsticks by issuing requests on its lt_i and rt_i ports. We assume that it always issues a request through its left chopstick before requesting the one on its right. Once both of its take requests are granted, it proceeds to eat for some time, at the end of which it then issues requests to free its left and right chopsticks by writing tokens on its lf_i and rf_i ports. A chopstick component has two input ports: t_i for take and f_i for free requests. Every chopstick is modeled by a FIFO1 channel and a synchronous drain [14]. The constraint automata for the interfaces of the philosophers and the chopsticks are shown in Figures 8.1(c) and (d). Note that the constraint automaton of a chopstick is obtained as a minimized product automaton of the constraint automata for FIFO1 and SyncDrain connectors.

Verification of the whole component-based system. The dining philosophers is an example of a system in which all components can be modeled by constraint automata without any internal action. It is a very restrictive assumption that all components can be modeled so. We need to consider a more general case: components are transition systems that have both internal and external actions and connectors are constraint automata all of whose actions are visible. In such cases, we can simply model any component by a labeled transition system and the coordinating system by a compositional constraint automaton. Labeled transition systems can be embedded in constraint automata as was shown in Lemma 8.3. The equivalence relations CFFD and NDFD are used to reduce the sizes of all constraint automata models and

then they are composed. Thus, the main steps of model checking of a complete component-based system, except the first, will be the same as we described for the coordination system. The first step is replaced with the following step and the other steps remain the same as the model checking algorithm for coordination subsystems:

1'- Model each component by an augmented labeled transition system and translate it to an augmented constraint automaton; and model the set of connectors directly by some augmented constraint automata.

Example 8.2 (A Resource Allocation System) As an example of a component based system, consider a resource allocation system with the requirement of mutual exclusion in using a resource as illustrated in Figure 8.2(a). The system consists of n processes which sometimes need to have access to a limited resource. The resource can be used by only one process at a time and it must be guaranteed that all requests for the resource are eventually granted. Also, if a process P has requested to use the resource, no other process is granted access to the resource more than once before the request of P has been granted. We suppose that each process has two ports: an output port rq through which it announces its request for using the resource and an input port gr through which the coordinator allows the process to access the resource and enter its critical section. The constraint automaton model of each process is shown in Figure 8.2(b). In this figure, state q_0 is the initial state of the process. In state q_1 the process announces its request for using the resource and waits for permission to access it. After receiving a signal gr, the process enters its critical section modeled by the state q_2. Once the process has finished using the resource, it turns the signal rq off and waits until the coordinator notices this and turns the signal qr off.

The coordination scenario performed by the coordinator to manage the resource is a pooling based allocation. For processes $P1$ to Pn sequentially, if rq_i is turned on, the coordinator turns the signal gr_i on and waits to observe the turning off of the signal rq_i. Then, it turns signal gr_i off. Figure 8.2(c) shows the constraint automaton model of the coordinator when there are two processes in the system. In this case, the last state P_4 is the same as the initial state P_0. This model can easily be generalized for the case of n processes.

This resource allocation system is an example of a component based system in which components (processes) are modeled by labeled transition systems, which can be embedded in constraint automata. In this case we model all internal actions by $\tau-$transitions. The coordinating subsystem is modeled directly by constraint automata without the need for $\tau-$transitions. In the next section, we report our results in compositional minimization of the resource allocation system using CFFD and NDFD equivalences.

Note that there are other compositional reasoning methods, such as the assumption-guarantee method [123], in which the reasoning is done separately on the components of the model by decomposing the desired property formula. Such techniques of compositional reasoning can be used in conjunction with our proposed minimization. We have not considered these methods here.

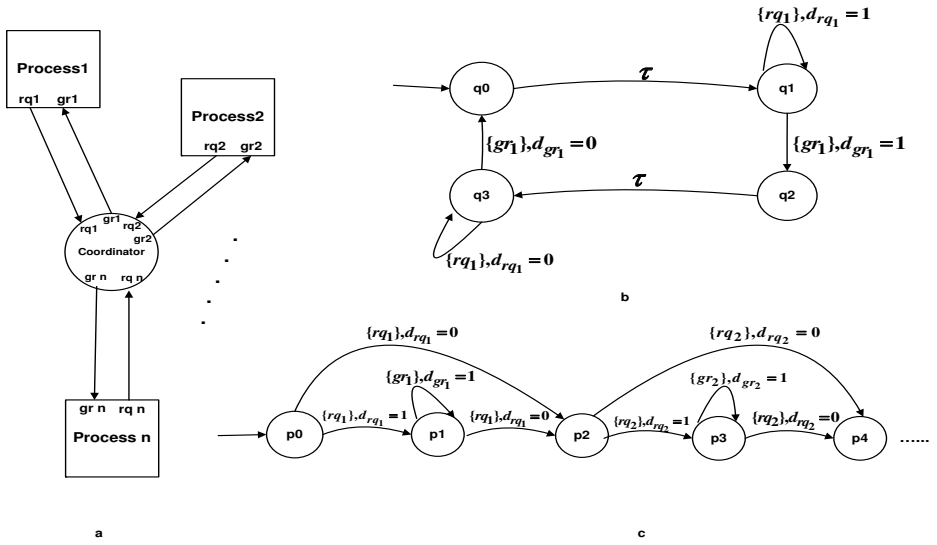


Figure 8.2: (a) A resource allocation system, (b) constraint automaton model of a process, (c) constraint automaton model of the coordinator

8.8 Case studies

We designed and implemented a tool for modeling and verification of systems modeled as constraint automata. One of the main goals of this tool is preparing an environment for specification of software architectures using constraint automata and verification of their properties, especially non-functional and qualitative properties of software architectures. Because of this aim the tool is called *ArQuVer* (Architecture Quality Verification tool). *ArQuVer* was implemented using Java. It receives the descriptions of a constraint automaton in XML format and can perform all composition operators defined in this thesis on them. It contains also components for minimizing constraint automata using (bi)simulation and CFFD equivalences. We can use its component for CFFD-minimization for NDFD-minimizing as well, through an intermediate software component.

Example 8.3 (Inres Protocol) As a case study, we considered the Inres protocol based system [63], as a component-based system whose coordinating subsystem can be modeled directly by a constraint automaton and its components can be modeled by labeled transition systems that can be transformed into constraint automata. The Inres protocol implements a reliable, connection oriented data transfer service, the *Inres service*, between two users. The main architecture of the protocol is shown in Figure 8.3. The Inres service is not symmetrical: it offers only a one way transition from an initiating process to a responding process. The

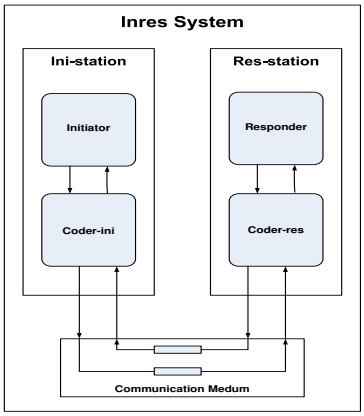


Figure 8.3: Inres protocol architecture (the connectors are Reo primitive channels)

protocol itself operates on top of a medium that offers a data transfer service. A description of the Inres protocol using the SDL language is presented in [54]. This description consists of four main processes: *Initiator* and *Responder* which implement the service by exchanging the protocol data units between themselves and *Coder-ini* and *Coder-res* which are used to hide the interface to the medium.

We modeled each of these four main processes by a labeled transition system, transformed into a constraint automaton, and considered the connectors between *Initiator* and *Coder-ini* and the connectors between *Responder* and *Coder-res* as two pairs of parallel Reo Sync channels. Also, we assumed that the low level medium of communication is a pair of parallel Reo $FIFOn$ channels, in which, n is a natural number constant. In the simplest case, $n = 1$. In an Inres system, components and connectors work and communicate in a sequential manner. First, *Initiator* activates the communication process, then its request is sent to *Coder-ini* through the Sync channel between them, and so on. We compose the models of all components and connectors with the join operator in the same order as the components and connectors are activated in the protocol. We hide the names of ports that can be considered as internal and invisible (in a more abstract view of the system). In the case study, we applied the compositional minimization method, namely, we minimized all constraint automata models before composing them. The minimizations were done using bisimulation, CFFD, and NDFD equivalences.

The results of our attempt to minimize the components of the Inres system are summarized in the columns A to E of Table 8.1 (more details in [115]). In Table 8.1, column A consists of the names of the components, column B contains the number of reachable states of the constraint automata models of the components without any minimization, column C reports the results after minimizing column B's models by bi-simulation relation, column D contains minimization of column B's models by CFFD, and column E contains minimization of column B's models by by NDFD. As we expect, the order of the sizes of the models in the columns B, C, D, and E is decreasing (except than for the size of the model of the commu-

A	B	C	D	E	F	G
Component name	CA, Not-reduced	Bisimulation	CFFD	NDFD	LTS, Not-reduced	Bisimulation
Initiator	145	34	30	24	131	30
Coder-ini	97	22	15	13	92	10
Responder	44	27	24	19	35	14
Coder-res	112	13	10	8	101	10
Ini-station	3451	1506	1145	1023	3217	1424
Res-station	1129	403	332	305	947	391
Comm. Medium	9	9	9	9	–	–
Inres System	164	54	36	30	135	28

Table 8.1: Number of reachable states for the Inres protocol system.

nication medium that is the same in all columns). It means that NDFD reduces models more than CFFD, and CFFD reduces more than bi-simulation relation. This fact is exactly because the bi-simulation equivalence relation preserves a bigger set of properties than CFFD. Also, CFFD preserves more properties than NDFD. See Proposition 8.11. For the case of the communication medium we have used the minimized model in all columns.

The main importance of our work is that it is the first attempt to model the Inres system by constraint automata and minimizing them using bi-simulation, CFFD, and NDFD equivalences and that (as a case study) it shows the applicability of failure based equivalences for model checking of constraint automata models. However, to be a more realistic and comparable case study, in columns F and G of Table 8.1, we summarize the results of another attempt to minimize the Inres system as reported in [105].

In the work reported in [105], all components have been modeled by labeled transition systems, the communication medium is considered as a set of peer-to-peer channels without any buffer (not modeled) and the minimization was done using only the bi-simulation equivalence relation. In Table 8.1, column F contains of the number of reachable states of the LTS models of the components without any minimization and column G reports the results after minimizing column F's models by bi-simulation relation, as reported in [105].

To compare the two works, first note that, we have modeled all components by constraint automata and minimized them using bi-simulation, CFFD, and NDFD equivalences. However, in the work reported in [105], the models are LTSs and only bi-simulation based minimization is considered. In our work, the order of minimizing and composing models of components is the same as reported in [105]. The sizes of our basic models of components (before minimization) are slightly bigger than the sizes reported in [105] because our models are constraint automata while in [105] simple labeled transition systems are used. Also, our models contain more details (less abstracted) and we modeled all connectors and channels while in [105] these are ignored. Because we don't have access to the exact models of the components used in [105], we can not give an indication of how many extra states,

No. of Processes	No Minimization	CFFD Minimization	NDFD Minimization
2	36	24	16
5	2430	60	40
10	1180980	120	80
20	5165606520	240	160
100	$>10^{49}$	1200	800

Table 8.2: Number of reachable states for the resource allocation system.

for instance, they would have if they modeled the system at the same level of detail as we do. Conversely, considering connectors (channels) as stateless models (as it is considered in [105]), is more abstract than we need to show the applicability and usefulness of our above mentioned method of model checking of the whole component-based systems including their coordination subsystems and components.

There is no well-established numerical relation between the sizes of models before and after minimization using one of the equivalence relations. For example, while the bi-simulation reduction from 135 to 28 in [105] produces a model 21% of its original size (see last row, columns F and G in Table 8.1), our bi-simulation reduction produces a model 33% of its original size. It is, of course, because that bi-simulation reduction is not linear. In fact, for another example, the situation can be in a reverse order. More importantly, in our results, the sizes of the reduced models using bi-simulation, CFFD and NDFD relations are about 33% (column C), 22% (column D), and 18% (column E) of the sizes of the original models, respectively. This shows the effectiveness of the reductions. As reported in [105], if one tries to generate the state space of the LTS model of the Inres protocol by the SDT Validator the final LTS model will have 388408 states and over 1880000 transitions. Because our constraint automata models contain more detail than their LTS models, the constraint automaton model of the Inres protocol will be bigger than its LTS model. Thus, our obtained minimizations using all three equivalence relations are highly significant.

Example 8.4 (Reduction of the resource allocation system models)

As a simple case study, we considered the compositional minimization method for the resource allocation system introduced in Example 8.2 using CFFD and NDFD equivalences. The results based on the number of processes in the system have been summarized in Table 8.2. Note that the structure of the model of the coordinating subsystem shown in Figure 8.2(c) is completely symmetric. Adding a new process to the system adds a block of two states with symmetric structures as the previous blocks to the automaton model of the coordinating subsystem. Further, the models of all processes are isomorphic. These symmetry and isomorphic facts give us the opportunity to construct the complete system using minimized basic models by a symmetric and repeating algorithm. These facts also result in a numerical relation between the number of processes and the number of reachable states in the minimized model of the system. In Table 8.2, we see that the numbers of reachable states of each minimized model using CFFD and NDFD equivalences are respectively 12 and 8 times the number of processes. This is a very interesting example that motivates using other kinds

of compositional verification methods, such as the *symmetric techniques* for model checking [56], in conjunction with the reduction techniques we proposed in this chapter, or with the automata theoretic techniques we proposed in the previous chapters.

9 Conclusions and Future Work

In this chapter, we conclude the presentation of our work in this thesis, summarize its results, and list topics for our future work.

9.1 Results and Conclusions

In this thesis, we presented a framework for automata theoretic model checking of coordination systems specified in Reo. As an operational modeling formalism that covers several intended behaviors of Reo connectors, such as fairness, I/O synchronization, and context dependency, we introduced Büchi automata of records (BAR) and their augmented version (ABAR). We showed that every constraint automaton (the first introduced operational semantics for Reo) can be translated into an essentially equivalent BAR. However, there are some Reo connectors whose behavior can be expressed in BAR or ABAR, but not in constraint automata.

To specify the properties to be verified, we introduced an action based linear temporal logic called ρ-LTL, interpreted over the executions of augmented Büchi automata of records. We showed how ρ-LTL formulas can be translated into their equivalent ABARs. The translation can be done inductively or using an on-the-fly method.

To deal with large state spaces, we showed that ABARs can be implemented using ordered binary decision diagrams (OBDD) as dense data structures. We described the implementation and case studies to show the applicability of our method to large state spaces.

We also showed that the state explosion problem can be tackled by a form of compositional minimization using some suitable equivalence relations. To this end, we proved that two failure based equivalence relations, called CFFD and NDFD, are congruence relations with respect to the join and hiding operators of constraint automata. These congruency results, together with the fact that CFFD and NDFD equivalences are minimal and preserve linear time temporal logic properties can be used for compositional minimization of constraint automata models in model checking. We showed the application of this method to some practical case studies.

9.2 Future Work

To continue the research presented in this thesis, in this section we list a number of topics that can be considered as future work. On the theoretical side, the following problems can be considered:

- Introducing *timed* versions of BARs and ABARs to be able to model real-time constraints with Reo connectors.

- Based on the above suggestion, introducing a timed version of the temporal logic ρLTL and its model checking, both globally and on-the-fly.

- Introducing *probabilistic* versions of BARs and ABARs to be able to model connectors with inherently probabilistic behaviors.
- Based on the above suggestion, introducing a probabilistic version of the temporal logic ρLTL and its model checking, both globally and on-the-fly.
- Introducing action based *branching time* temporal logics for BAR and ABAR models.
- Based on the above suggestion, investigating the model checking of branching time properties of connectors modeled by BAR and ABAR.
- The branching time case can also be considered for timed BAR and ABAR and their model checking.
- The branching time case can also be considered for probabilistic BAR and ABAR and their model checking.
- The results of this thesis can be focused in particular for some more practical fields of software engineering such as *software quality measurement*, *service-oriented models* of software, and several other *non-functional* properties.
- Some other methods to deal with the state explosion problem seem to be very suitable for the case of Reo nets modeled by BAR and ABAR or by constraint automata. We suggest considering the methods of *abstraction*, *symmetry*, and *assume-guarantee based compositional reasoning*.
- The method of compositional minimization introduced in thesis was based on constraint automata. Using this method for BAR and ABAR models seems to be more realistic and achievable. This can be investigated in the future. To this end we need the following theoretical results:
  - Proving that the failure based equivalences CFFD and NDFD are congruences with respect to all composition operators of BARs and ABARs.
  - Proving that CFFD and NDFD preserve sets of linear temporal properties interpreted over BAR and ABAR models, and that they are the weakest congruences that satisfy the preservation of these properties.
  - Introducing minimization algorithms for BAR and ABAR models using CFFD and NDFD equivalences.

We intend to enhance our tool, especially by incorporating the global and on-the-fly translations of ρLTL formulas into augmented Büchi automata of records that we introduced in this thesis. Moreover, we plan to integrate our BDD-based model checker and our tool for compositional minimization of constraint automata in our tool set. Finally, we will integrate our tool set within the Extensible Coordination Tools [2] programming environment for Reo.

Bibliography

[1] *CADP Toolset User Manual*, Available through `http://www.inrialpes.fr/vasy/cadp/man`.

[2] *Extensible Coordination Tools Homepage*, Available through `http://www.eclipse.org`.

[3] *JINC, A BDD library*, Available through `http://www.jossowski.de`

[4] *NuSMV Model Checker User Manual*, Available through `http://nusmv.irst.itc.it/NuSMV`.

[5] Reo Homepage, Available through `http://reo.project.cwi.nl/`

[6] *Spin Model Checker Manual*, Available through `http://netlib.bell-labs.com/netlib/Spin`.

[7] *Vereofy Model Checker User Manual*, Available through `http://www.vereofy.de/`.

[8] Ahmadi D. Z., **Izadi M.**, *Modeling Real-Time Coordination Systems Using Timed Büchi Automata*, Proc. of CSI Int. Symp. on Computer Science and Software Engineering (CSSE2011), IEEE Xplore Digital Library, pp. 17-24, (2011).

[9] Ahuja S., Carriero N., Gelernter D., *Linda and Friends*, IEEE Computer, **19 (8)**, pp. 26-34, (1986).

[10] de Alfaro L., Henzinger T.A., *Interface Automata*, Proc. of the 9th Annual ACM Symp. on Foundations of Software Engineering (FSE 2001), pp. 109-120, (2001).

[11] Alpern B., Schneider F. B., *Defining liveness*, Information Processing Letters **21**, (1985), pp. 181 185.

[12] Arbab F., *What Do You Mean, Coordination?*, Bulletin of the Dutch Association for Theoretical Computer Science, (1998).

[13] Arbab F., *Reo: A Channel-based Coordination Model for Component Composition*, Math. Struc. in Computer Science, **14(3)**, (2004), 329-366.

[14] Arbab F., *Abstract Behaiviour Types: A foundation model for components and their composition*, science of Computer Programming, **55**, (2005), 3-52.

[15] Arbab F., *Composition of Interacting Computations*, in *Interactive Computation: The New Paradigm*, D. Goldin, S. Smolka, and P. Wegner (Eds.), Springer-Verlag, (2006).

[16] Arbab F., *A Behavioral Model for Composition of Software Components*, L'Objet, Lavoisier, Vol. 12, No. 1, pp. 33-76, (2006).

[17] Arbab F., Baier C., de Boer F., Rutten J., *Models and Temporal Logics of Timed Component Connectors*, Proceedings of SEFM 2004, pp. 198-207, IEEE CS Press, (2004).

[18] Arbab F., Baier C., de Boer F., Rutten J., *Models and Temporal Logical Specifications for Timed Component Connectors*, Software and System Modeling, **6(1)**, pp. 59-82, Springer, (2007).

[19] Arbab F., Baier C., de Boer F., Rutten J., Sirjani M., *Synthesis of Reo circuites for implementation of component-connector automata specifications*, Proceedings of CORDINATION 2005, LNCS, **3454**, Springer-Verlag, (2005), 236-251.

[20] Arbab F., de Boer F., Bonsangue M., Guillen Scholten J., *A Channel-Based Coordination Model for Components*, CWI Report SEN-R0127, (2001),

[21] Arbab F., Bruni R., Clarke D., Lanese I, Montanari U., *Tiles for Reo*, In *Recent Trends in Algebraic Development Techniques*, volume 5486 of LNCS, pages 3755. Springer, 2009.

[22] Arbab F., Chothia T., van der Mei R., Meng S., Moon Y. J., Verhoef C., *From Coordination to Stochastic Models of QoS*, In John Field and Vasco Vasconcelos, editors, Coordination Models and Languages, LNCS **5521**, pp. 268-287. Springer, (2009).

[23] Arbab F., Chothia T., Meng S., and Moon Y. J., *Component Connectors with QoS Guarantees*. In *Coordination Languages and Models: Proc. Coordination 2007*, Lecture Notes in Computer Science, Springer-Verlag, 2007.

[24] Arbab F., Herman I., Spilling P., *An overview of Manifold and its implementation*, in Concurrency - Practice and Experience **5(1)**, pp. 23-70, (1993).

[25] Arbab F., Mavadat F., *Coordination Through Channel Composition*, Proceedings of Coordination Languages and Models 2002, LNCS, **2315**, Springer-Verlag, (2002).

[26] Arbab F., Meng S., Moon Y. J., Kwiatkowska M., Qu H., *Reo2MC: a Tool Chain for Performance Analysis of Coordination Models*, In Proceedings of ESEC/FSE 2009, pp. 287-288, (2009).

[27] Arbab F., Rutten J.J.M.M., *A Coinductive Calculus of Componnent Connectors*, Proc. of the 16th International Workshop on Algebraic Development Techniques (WADT 2002), M. Wirsing, D. Pattinson and R. Hennicker (eds.), LNCS, **2755**, Springer-Verlag, (2003), 34-55.

[28] Baier C., *Probabilistic Models for Reo Connector Circuits*, Journal of Universal Computer Science, **11(10)**, pp. 1718-1748, (2005).

[29] Baier C., Katon J.P., *Principles of Model Checking*, The MIT Press, (2008).

[30] Baier C., Sirjani M., Arbab F., Rutten J., *Modelling Component connectors in Reo by Constraint Automata*, Science of Computer Programming, **61**, pp. 75-113, (2006).

[31] Baier, C., and Wolf, V. *Stochastic reasoning about channel-based component connectors*. In *Coordination Languages and Models: Proc. Coordination 2006*, Lecture Notes in Computer Science, Springer-Verlag, 2006.

[32] Barbosa M., Barbosa L., *A perspective on service orchestration*, Science of Computer Programming, 74(9), pp. 671687, Elsevier, 2009.

[33] Bliudze, S. and Sifakis, J. *The Algebra of Connectors - Structuring Interaction in BIP*. IEEE Transactions on Computers, **57:10**, IEEE Computer Society (2008), 1315-1330.

[34] Bollig B., Wegener I., *Improving the variable ordering of OBDDs is NPcomplete*, IEEE Transactions on Computers, 45(9):9931002, (1996).

[35] Bolognesi T., Caneve M., *Equivalence Verification: Theory, Algorithms and a Tool*, in The Formal Description Technique LOTOS, North-Holand, (1989), 303-326.

[36] Bonsangue M., Clarke D., Silva A., *A model of context-dependent component connectors*, To appear in *Science of Computer Programming*, special issue dedicated to Coordination2009, Elsevier, (2011).

[37] Bonsangue, M., Clarke D., Silva A., *Automata for context-dependent connectors*. In Field J., Vasconcelos V.T. (eds.) LNCS, vol. 5521, pp. 184-203. Springer (2009).

[38] Bonsangue M.M., **Izadi M.**, *Automata Based Model Checking for Reo Connectors*, In proceedings of FSEN 2009, Lecture Notes in Computer science (LNCS) 5961 Springer 2010.

[39] Bruni R., Fiadeiro J., Lanese I., Lopez A. and Montanari. U., *New Insights on Architectural Connectors* , in: Levy J.-J., Mayr E.W., and Mitchell J.C., Eds., "Proceedings of IFIP TCS 2004", 3rd IFIP International Conference on Theoretical Computer Science, Kluwer Academics, (2004), 367-379.

[40] Bryant R., *Graph-based algorithms for boolean function manipulation*, in IEEE Transactions on Computers, **35(8)**, pp. 677691, (1986).

[41] Carriero N., Gelernter D., *Linda in Context*, Communications of the ACM, **32 (4)**, pp. 444-458, (1989).

[42] Carriero N., Gelernter D., *Coordination Languages and their Significance*, Communications of the ACM, **35 (2)**, pp. 97-107, (1992).

[43] Cimatti A., Clarke E.M., Giunchiglia E., Giunchiglia F., Pistore M., Roveri M., Sebastiani R., Tacchella A., *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*, Proceedings of the 14th CAV, Springer's LNCS **2404**, (2002), 359-364.

[44] Clarke D., *Reasoning about Connectors Reconfoguration I: Equivalence of Constructions*, CWI Technical Report SEN-R0506, (2004).

[45] Clarke D., *Reasoning about Connectors Reconfoguration II: Basic Reconfiguration Logic*, Proceedings of FSEN05, Tehran, Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier (2005).

[46] Clarke D., *Coordination: Reo, nets, and logic*. In F.S. de Boer, M.M. Bonsangue, S. Graf, and W.-P. de Roever (eds) *Sixth International Symposium on Formal Methods on Components and Objects (FMCO 2008) – State-of-the-Art Survey*, volume 5382 of *Lecture Notes in Computer Science*, pages 226–256, Springer, 2008.

[47] Clarke D., Costa D., and Arbab F., *Connector colouring I: synchronisation and context dependency*, Science of Computer Programming, **66(3)**, (2007), 205-225.

[48] Clarke E., Emerson A., Sistla A., *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems **8(2)**, pp. 244-263, (1986).

[49] Clarke E., Grumberg O., Long D., *Model Checking and Abstraction*, ACM Transactions on Programming Languages and Systems, **16(5)**, (1994), 1512-1542.

[50] Clarke E., Grumberg O., Peled D., "Model Checking", The MIT Press, 1999.

[51] Clarke E., Long D., McMillan K., *Compositional Model Checking*, Proceeding of the 4th IEEE Symposium on Logic in Computer Science, (1989), 353-362.

[52] Costa D., *Formal Models For Component Connectors,* Ph.D. thesis, VUA (2010).

[53] Cronkovic I., Hnich B., Jonsson T., Kiziltan Z., *Specification, Implementation, and Deployment of Components*, Communications of the ACM, **Vol.45, No.10**, (2002), 35-40.

[54] Ellesberger J., Hogerfe D., Sarma A., *SDL Formal Object Oriented Language for Communicating systems*, Prentice Hall, (1997).

[55] Emerson E. A., *Temporal and modal logic*, In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, vol B: Formal Models and Semantics, Elsevier Publishers B.V., (1990).

[56] Emerson A., Sistla A., *Symmetry and Model Checking*, Proceedings of CAV'93, (1993), 463-478.

[57] Emerson EA. and Lei CL, *Efficient model checking in fragments of the propositional mu-calculus,* IEEE Computer Society Press, 1986.

[58] Gastin, P., and Oddoux D., *Fast LTL to Büchi Automata Translation*, Proceedings of the 13th International Conference on Computer Aided Verification CAV01, LNCS, vol. 2102, pp. 53-65, Springer-Verlag (2001).

[59] Gerth R., Peled D., Vardi M., Wolper P., *Simple On-the-fly Automatic Verification of Linear Temporal Logic*, Proc. IFIP-WG6.1 Symp. Protocol Specification, Testing, and Verification (PSTV95), pp. 3-18, Warsaw, Poland, Chapman & Hall, June 1995.

[60] Graf S., Steffen B., *Compositional Minimization of Finite-State Systems*, Proceedings of CAV'90, Springer-Verlag, (1991), 186-196.

[61] Groote J. F., Mathijssen A. H. J., Reniers M. A., Usenko Y. S., and van Weerdenburg M. J., *The formal specification language mCRL2*, In Methods for Modelling Software Systems, IBFI, Schloss Dagstuhl, (2007).

[62] Hoare C.A.R., "Communicating Sequential Processes", Prentice-hall, (1985).

[63] Hogrefe D., *OSI formal specification case study: the Inres protocol and service*, Technical Report IAM-91-012, Inst. fur Informatik, Universitut Bern, (1991).

[64] Hojati R., Touati H., Kurshan R., and Brayton R., *Efficient ω-regular language containment,* Computer Aided Verification, pp. 396-409, 1993, Springer.

[65] Holzmann G.J., *The Model Checker SPIN*, IEEE Transactions on software engineering, **23(5)**, (1997), 279-295.

[66] Hopcroft J.E., Motwani R., Ullman J.D., *Introduction to Automata Theory, Languages, and Computation* , 3rd edition, Addison-Wesley (2006).

[67] Hromkovic J., "Algorithmics for hard problems", Springer, (2001).

[68] Huth M., Ryan M., *Logic in Computer Science: Modelling and Reasoning about Systems*, second edition, Cambridge University Press, (2004).

[69] **Izadi M.**, *An Integrated Formal Method for Specification and Verification of Component-Based Syatems*, PhD Thesis of Computer Software Engineering, Dept. of Computer Engineering, Sharif University of Technology, Tehran, Iran, (2008).

[70] **Izadi M.**, *Typed Temporal Logic: A General Framework for Verification of Non-functional and Security Requirements of Component Based Systems*, Proceedings of seventh school in MOdeling and VErification of Parallel Programs (MOVEP06), Bordeaux, France, (2006), 305-311.

[71] **Izadi M.**, Bonsangue M.M., *Recasting constraint automata into Büchi automata*, in Proc. ICTAC 2008, Lecture Notes in Computer Science **5160**, Springer-Verlag, pp. 156-170, (2008).

[72] **Izadi M.**, Bonsangue M.M., Clarke D., *Büchi automata for modeling component connectors*, in Journal of Software and System Modelling, **10(2)**, pp.183-200, Springer-Verlag, (2011).

[73] **Izadi M.**, Bonsangue M.M., Clarke D., *Modeling Component Connectors: Synchronisation and Context-Dependency*, in Proc. 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2008), pp. 303-312, (2008).

[74] **Izadi M.**, Movaghar A., *Failure-based equivalence of constraint automata*, International Journal of Compututer Mathematics, **87(11)**, pp. 2426-2443, (2010).

[75] **Izadi M.**, Movaghar A., *Compositional Failure-based Equivalence of Constraint Automata*, Electr. Notes in Theor. Comput. Sci., **250(1)**, pp. 105-122, Elsevier (2009).

[76] **Izadi M.**, Movaghar A., *Compositional Model checking of Component Based Software Using Compositional Reductions*, International Journal of Software Engineering and Knowledge Engineering (IJSEKE) **18(5)**, WorldScientific Pub. Co., (August 2008).

[77] **Izadi M.**, Movaghar A., Arbab F., *Model Checking of Component Connectors*, proc. of COMPSAC2007, pp.673-675, IEEE Computer Society Press, (2007).

[78] **Izadi M.**, Movaghar A., *Compositional failure-based semantic equivalences for Reo specifications*, proc. SAVCBS2007, pp.99-100, (2007).

[79] **Izadi M.**, Movaghar A., *An Equivalence Based Method for Compositional Verification of the Linear Temporal Logic of Constraint Automata*, Proceedings of FSEN05, Electronic Notes in Theoretical Computer Science (ENTCS), **159**, pp.171-186 , Elsevier (2006).

[80] **Izadi M.**, Movaghar A., *An Efficient Model Checking Algorithm for a Fragment of μ-Calculus*, CSI Journal of Computer Science and Engineering (JCSE), Vol. 3, No. 3(a), pp.43-53, Fall (2005).

[81] **Izadi M.**, Movaghar A., *An Efficient Model Checking Algorithm for a Fragment of μ-Calculus*, Proceedings of the Seventeenth International Conference on Software Engineering and Knowledge Engineering (SEKE2005), Taiwan, July 14 to 16, (2005).

[82] **Izadi M.**, Movaghar A., *A Formal System for Compositional and Hierarchal Modeling and Verification of Component Based Computing Systems*, Proceedings of the International Symposium in Telecommunication 2005 (IST2005), Iran Telecommunication Research Center (ITRC), Shiraz, Iran, (2005).

[83] Jongmans S. S. T. Q., Krause c., Arbab A., *Encoding Context-Sensitivity in Reo into Non-Context-Sensitive Semantic Models*, In Proceedings of COORDINATION2011, pp. 31-48, (2011).

[84] Jongmans S. S. T. Q., Arbab A., *Semantic Models of Connectors: A Study on Equivalence*, In Proceedings of the 4th International Workshop on Interaction and Concurrency Experience (ICE 2011), satellilte event of DisCoTec (2011).

[85] Kaplan D.M., *Regular expressions and the equivalence of programs*, Journal of Compiting System Science, **3**, (1969) 361-386.

[86] Kaivola R., Valmari, A., *The Weakest Semantic Equivalence Preserving Nexttime-less Linear Temporal Logic*, Proceedings of CONCUR'92, LNCS, **630**, Springer-Verlag, (1992), 207-221.

[87] Kanters O., Verhoef C., Schut M., *QoS analysis by simulation in Reo*, Vrije Universiteit Amsterdam, The Netherlands (2010).

[88] Kemper S., *SAT-based Verification for Timed Component Connectors*, Electr. Notes Theor. Comput. Sci., **255**, pp. 103-118, (2009).

[89] Kemper S., *Compositional construction of real-time dataflow networks*, In Dave Clarke and Gul A. Agha editors, Proceedings of COORDINATION 2010, LNCS, **6116**, pp. 92-106. Springer-Verlag, (2010).

[90] Koehler C., Clarke D., *Decomposing port automata*, In SAC'09 Proc., 2009 ACM Symposium on Applied Computing, pp. 13691373, New York, USA, (2009).

[91] Kokash N., Krause C., de Vink E.P., *Reo + mCRL2: A Framework for Model-checking Dataflow in Service Compositions*, Formal Aspects of Computing 2011, Springer-Verlag, will appear.

[92] Kokash, N., Krause, C., de Vink, E.P., *Time and Data Aware Analysis of Graphical Service Models in Reo*, IEEE International Conference on Software Engineering and Formal Methods (SEFM'10), IEEE Computer Society, pp. 125-134, (2010).

[93] Kokash N., Krause C., de Vink E.P., *Data-Aware Design and Verification of Service Compositions with Reo and mCRL2*, Proceedings of the ACM Symposium on Applied Computing, Technical track on Service Oriented Architectures and Programming, March 2010, Sierre, Switzerland, pp. 2406-2413, (2010).

[94] Kokash N., Krause C., de Vink E.P., *Verification of Context-Dependent Channel-Based Service Models*, International Symposium on Formal Methods for Components and Objects (FMCO'09), LNCS **6286**, Springer, pp. 21-40, (2009).

[95] Krause C., *Reconfigurable Component Connectors*, PhD Thesis, Leiden University, (2011).

[96] Krause C., *Distributed port automata*, In 10th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'11), Electronic Communications of the EASST (to appear), (2011).

[97] Kwiatkowska, M. Z., *Survey of fairness notions*, Information and Software Technology, **31**, (1989), pp. 371-386.

[98] Kluppelholz S., Baier C., *Symbolic Model Checking for Channel-based Component Connectors*, Science of Computer Programming, **74(9)**, pp. 688-701, Elsevier (2009).

[99] Kluppelholz S., Baier C., *Symbolic Model Checking for Channel-based Component Connectors*, Proceedings of FOCLASA2006, Elsevier's ENTCS, **175(2)**, pp. 19-37, Elsevier, (2007).

[100] Kozen D., *Automata on guarded strings and applications*, Matématica Contemporânea, **24** (2003), 117-139.

[101] Kupferman O., Vardi M., *Verification of Fair Transition Systems*, Proceedings of the Eighth International Conference on Computer Aided Verification CAV, (1996).

[102] Kurshan R. P., "Computer-aided Verifcation of Coordinating Processes: The Automata-Theoretic Approach", Princeton University Press, (1994).

[103] Lamport L., *What Good is Temporal Logic?*, Information Processing, **83**, Elsevier, (1983), 657-668.

[104] Loiseaux C., Graf S., Sifakis J., Bouajjani A., and Bensalem S., *Property preserving abstractions for the verifcation of concurrent systems*, Formal Methods in System Design, 6(1), pp.11-44, (1995).

[105] Luukkainen M., Ahtiainen A., *Compositional Verification of SDL Descriptions*, Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC (SAM'98), (1998).

[106] Lynch N., "Distributed Algorithms", Morgan Kaufman Publishers, (1996).

[107] Madhusudan, P. *On the fly model checking for linear time temporal logic*, M.Sc. Thesis (1996), Anna University, Madras, India.

[108] Makarem M.A., "Formal Specification and Verification of Software Architecture Properties", M.Sc. degree Dissertation, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran, 2006.

[109] Makarem M.A., Mirian S.H., "Formal Modeling and Verification of Software Architecture Quality Attributes", Proceedings of CSICC 2006, Institute for Theoretical Phisycs and Matematics (IPM), Tehran, pp. 455-466, (2006).

[110] Manna Z., Pnueli A., "The Temporal Logic of Reactive and Concurrent Systems: Specification", Springer-Verlag, (1991).

[111] McMillan K. L., "Symbolic Model Checking An Approach to the State Explosion Problem", Kluwer Academic, (1993).

[112] Milner R., *Communication and Concurrency*, Prentice-Hall, (1989).

[113] Moon Y. j., *Stochastic Models for Quality of Service of Component Connectors*, PhD thesis, Leiden University, (2011).

[114] Moon Y.J., Silva A., Krause C., Arbab F., *A Compositional Semantics for Stochastic Reo Connectors*, In Proc. International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2010), EPTCS 30, pp. 93107, (2010).

[115] Movaghar A., **Izadi M.**, *Compositional Verification of Temporal Logic Specifications*, Proceedings of Research Reports of Sharif University of Technology, (2006), 203-224.

[116] Mousavi M.R., Sirjani M., and Arbab F., *Formal sematics and analysis of component connectors in Reo*, in Proc. of FOCLASA 2005, Elsevier's ENTCS **154**, (2005), 83-99.

[117] Muller O., "A Verification Environment for I/O Automata Based on Formalized MetaTheory ," PhD thesis, Institut fur Informatik, Technische Universitut Munchen, (1998).

[118] NavidPour S., **Izadi M.**, *Linear Temporal Logic of Constraint Automata*, Communications in Computer and Information Science, 6(2), pp. 972-975, Springer (2009).

[119] NavidPour S., **Izadi M.**, Movaghar A., *Live and Fair Constraint Automata and Their Linear Temporal Logic of Steps*, proc. of COMPSAC2008, pp.211-218, (2008).

[120] Papadopoulos G. A., Arbab F., *Coordination Models and Languages*, Advances in Computers **46**, Academic Press, 1998.

[121] Peled D., *Verification for Robust Specification*, Conference on Theorm Proving in Higher Order Logic, Springer-Verlag, (1997), 231-241.

[122] PeykAsa Company, *Overall Architecture of High Troughput Short Massage Service Center*, Technical Report PA-HTSMSC 830510-r3, PeykAsa Massageware, Tehran, Iran (2005).

[123] Pnueli A., *In Transition from Global to Modular Temporal Reasoning about Programs*, "Logics and Models of Concurrent Systems", NATO ASI series, **F13**, Springer-Verlag, (1985), 123-146.

[124] Pradella M., San Pietro P, Spoletini P, and Morzenti A., Practical model checking of LTL with past,ATVA03: 1st Workshop on Automated Technology for Verification and Analysis, (2003).

[125] Proena J., *Deployment of Distributed Component Based Systems*, PhD thesis, Leiden University, The Netherlands, (2011).

[126] Ravi K., Bloem R., and Somenzi F., *A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles*, Formal Methods in Computer-Aided Design, pp. 162-179, Springer, (2000).

[127] Remy D., *Efficient representation of extensible records*, Proc. ACM SIGPLAN Workshop on ML and its applications, (1994), 12-16.

[128] de Roever W. P., Langmaack h., Pnueli A., *Compositionality: The Significant Difference*, International Symposium, COMPOS'97, Bad Malente, Germany, September 1997, Revised Lectures, Lecture Notes in Computer Science, **1536**, Springer-Verlag, (1998).

[129] Romijn J., Vaandrager F., *A note on fairness in I O automata*, Information Processing Letters, **59(5)**, (1996), pp. 245-250.

[130] Safra S., *On the complexity of ω-automata*, Lectures, Proceedings of 29th IEEE FOCS, (1988), 319-327.

[131] Schneider k., *Verification of Recative Systems*, Springer, (2004).

[132] Schnoebelen P., *The Complexity of Temporal Logic Model Checking*, Advances in Modal Logic **4**, World Scientific Publishing Co., (2002), 1-44.

[133] Scholten J.G., Arbab F., de Boer F., Bonsangue, *A Channel-based Coordination Model for Components*, Electr. Notes Theor. Comput. Sci., **68(3)**, (2003).

[134] Sistla A.P., Clarke E.M., *The Complexity of Propositional Linear Temporal Logic*, Journal of the ACM **32**, (1985), 733-749.

[135] Szyperski C., "Component Software: Beyond Object-Oriented Programming", first edition, Addison-Wesley, (1998).

[136] Szyperski C., Gruntz D., and Murer S. *Component software: beyond object-oriented programming (second edition)*, Addison-Wesley, 2002.

[137] Tarjan R., *Depth first search and linear graph algorithms*, SIAM Journal of Computing 1, pp. 146160, (1972).

[138] Thomas W., *Automata on Infinite Objects*, J. van Leeuwen (editor), "Handbook of Theoretical Computer Science", **vol. B**, Elsevier, (1990), 133-191.

[139] Valmari A., *The State Explosion Problem*, Lectures on Petri Nets I: Basic Models, LNCS **1491**, Sprnger-Verlag, (1998), 429-529.

[140] Valmari A., *Failur-based Equivalences are Faster than Many Believe*, Proc. Structures in Concurrency Theory, May 1995, Springer-Verlag (1995), 326-340.

[141] Valmari A., Tienari M., *An Improved Failure Equivalence for Finite State Systems with a Reduction Algorithm*, "Protocol Specification, Testing and Verification", **XI**, (1991), 3-18.

[142] Valmari A., Tienari M., *Compositional Failure Based Semantic Models for Basic LOTOS*, Formal Aspects of Computing **7**, (1995), 440-468.

[143] van Glabbeek R.J., *The Linear Time - Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes*, In J. Bergstra, A. Ponse and S. Smolka (editors), Handbook of Process Algebra, chapter 1, Elsevier Science, (2001), 3-99.

[144] van Glabbeek R.J., *The Linear Time - Branching Time Spectrum II: The semantics of sequential systems with silent moves*, LNCS **715**, Springer-Verlag, (1993), 66-81.

[145] Vardi M., *An Automata-Theoretic Approach to Linear Temporal Logic*, Lecture Notes in Computer Science, **1043**, Springer-Verlag (1996), 238-266.

[146] Vardi M., *Automata-The oretic Model Checking Revisited*, Proceedings of 8th Conference on Verification, Model Checking, and Abstract Interpretation, Lecture Notes in Computer Science, **4349**, Springer-Verlag (2007), 137-150.

[147] Vardi M., *Branching vs. Linear Time: Final Showdown*, in proc. of TACAS2001, pp. 1-22, (2001).

[148] Vardi M., *Linear vs. Branching Time: A Complexity Theoretic Perspective*, Electr. Notes Theor. Comput. Sci. **68(4)**, (2002).

[149] Vardi M., Wolper, P. *An Automata-Theoretic Approach to Automatic Program Verification*, in Proc. 1st. symposium on Logic in Computer Science, (1986), 322-331.

[150] Verhoef C., Krause C., Kanters O., van der Mei R., *Simulation-Based Performance Analysis of Channel-Based Coordination Models*, In Wolfgang de Meuter and Gruia-Catalin Roman, editors, Coordination Models and Languages, LNCS **6721**, pp. 187-201, Springer, (2011).

[151] Virtanen H., Hansen H., ValmariA., Nieminen J., Erkkil T., *Tampere Verification Tool*, Tools and Algorithms for the Construction and Analysis of Systems, LNCS **2988**, Sprnger-Verlag, (2004), 153-157.

[152] Wolper P., *Temporal Logic Can be More Expressive*, Information and Control, **56**, (1983), 72-99.

A Abstract

In this thesis, we present a framework for automata theoretic model checking of coordination systems specified in Reo. Reo is a coordination language that is based on a calculus of channel composition. Using Reo specifications, complex connectors can be built compositionally, organized as a network of channels to interconnect and orchestrate or choreograph the interactions among a set of concurrent components and/or distributed services.

We introduce Büchi automata of records (BAR) and their augmented version (ABAR) as an operational modeling formalism that covers several intended forms of behavior of Reo connectors, such as fairness, I/O synchronization, and context dependency. To specify the properties to be verified, we introduce an action based linear temporal logic, called ρ-LTL, interpreted over the executions of augmented Büchi automata of records, and show how the formulas can be translated into ABARs. This translation can be done either inductively, or by using an on-the-fly method. To deal with the large state spaces, we show that ABARs can be implemented using ordered binary decision diagrams (OBDD). For this purpose, we also introduce the necessary modifications over the basic model checking algorithm that can be applied directly over OBDD structures. Our implementation and a number of case studies that we carried out show the applicability of our method over large state spaces.

We also show that the state explosion problem can be tackled by compositional minimization methods using some suitable equivalence relations. In fact, we show two equivalences that are congruences with respect to the connector composition operators and such that they both preserves linear time temporal logic properties. Again, we demonstrate our method by means of few practical case studies.

B Samenvatting (dutch)

In dit proefschrift presenteren we een framewerk voor het automaat-theoretisch model checken van cordinatiesystemen gespecificeerd in Reo. Reo is een cordinatietaal gebaseerd op een calculus van kanaalcompositie. Met behulp van Reo specificaties kunnen complexe connectoren compositioneel gebouwd worden, georganiseerd als een netwerk van kanalen, om een verzameling van concurrente componenten en/of gedistribueerde diensten te verbinden en orkestreren, of hun interactie te choreo-graferen.

We introduceren Büchi automaten van records (BAR) en hun uitgebreide versie (ABAR) als een operationeel modelleringsformalisme dat verschillende gedragsvormen van Reo connectoren beslaat, zoals fairness, I/O-synchronisatie, en context-afhankelijkheid. Om de te verifiren eigenschappen te specificeren introduceren we een op actie gebaseerde lineaire temporele logica, genaamd ρ-LTL, genterpreteerd over de executies van uitgebreide Bchi automaten van records, en laten zien hoe de formules vertaald kunnen worden naar ABARs. Deze vertaling kan inductief worden uitgevoerd, of met een on-the-fly methode. Om te kunnen omgaan met een grote toestandsruimte, laten we zien dat ABARs gemplementeerd kunnen worden met geordende binaire beslissingsdiagrammen (OBBDs). Hiervoor introduceren we ook de nodige aanpassingen op het basis model checking algoritme dat direct op OBDD structuren toegepast kan worden. Onze implementatie, en een bepaald aantal case-studies die we hebben uitgevoerd, laten zien dat onze methode toepasbaar is op systemen met een grote aantal toestanden.

We laten ook zien dat het toestand explosie probleem aangepakt kan worden met compositionele minimalisatiemethoden, met behulp van geschikte equivalentierelaties. Hiervoor laten we zien dat twee equivalentierelaties die congruenties zijn met betrekking tot connector compositie operatoren en zodanig dat ze beiden behouden lineaire tijd temporele logica eigenschappen. Nogmaals, we hebben deze methode gedemonstreerd aan de hand van enkele praktische case studies.

C Curriculum Vitae

Mohammad Izadi was born in Najafabad, Iran, on December 23, 1972. He finished his high school education in his hometown with two distinguished diplomas, one in mathematics and the other in humanities. This is the beginning of his simultaneous educations and academic jobs in both humanities (in particulars, philosophy) and natural sciences (in particular, computer science).

He moved to Tehran in 1990, where he started his four years undergraduate studies in computer hardware engineering at Sharif University of Technology. In 1995, he was accepted at Sharif University of Technology for Master's study in philosophy of science. He finished his Master's in 1997 with a distinguished dissertation that was nominated as the best dissertation of the university in basic sciences for a competition in the Iranian Ministry of science. In 1998, he was employed as a research assistant in Iranian Academy of Philosophy and later as a faculty member in 2000. From September 2000 to December 2008 he received his second Master's degree and a PhD degree in computer software engineering from Sharif University of Technology. From December 2008 to September 2011, Mohammad was an assistant professor at the Research Institute for Humanities and Cultural Studies (IHCS, a new name for the above mentioned academy) in Tehran and also an adjunct assistant professor of computer engineering at Sharif University of Technology. Since September 2011, he is a tenure faculty member of the department of computer engineering at Sharif University of Technology and adjunct assistant professor of philosophy at IHCS.

From January 2008 to December 2011, he has been a PhD candidate at the Leiden Institute of Advanced Computer Science (LIACS) of Leiden University, in the Netherlands.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks:*

energy-efficient attack and defense. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*.

Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22