

Electronic Notes in Theoretical Computer Science 250 (2009) 105–122

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Compositional Failure-based Equivalence of Constraint Automata

# Mohammad Izadi[1](#_bookmark0)

*Department of computer Engineering, Sharif University of Technology, Tehran, Iran.*

*Research Institute for Humanities and Cultural Studies (IHCS), Tehran, Iran.*

*Leiden Institute of Advanced Computer Science (LIACS), Leiden University, The Netherlands.*

# Ali Movaghar[2](#_bookmark0)

*Department of computer Engineering, Sharif University of Technology, Tehran, Iran.*

Abstract

Reo is a coordination language for modeling component connectors of component-based computing sys- tems. Constraint automaton, as an extension of finite automaton, has been proposed as the operational semantics of Reo. In this paper, we introduce an extended definition of constraint automaton by which, every constraint automaton can be considered as a labeled transition system and each labeled transition system can be translated into a constraint automaton. We show that failure-based equivalences CFFD and NDFD are congruences with respect to composition of constraint automata using their join (production) and hiding operators. Based on these congruency results and by considering the temporal logic preservation properties of CFFD and NDFD equivalences, they can be used for reducing sizes of models before doing model checking based verification.

*Keywords:* Constraint Automata, Failure-based Equivalences, Component-based Systems, Compositional Model Checking, Equivalence-based Reduction.

# Introduction

The concept of *component based systems*, especially component based software, is a philosophy or way of thinking to deal with the complexity in designing large scale computing systems. One of the main goals of this approach is to compose reusable components by some glue codes. The model or the way in which these components are composed is called *coordination model*. Sometimes there are some formal or programming languages which are used for specification of coordination models. Such languages are called as *coordination languages*. Reo, as one of the

1 Email: [Izadi@ce.sharif.edu](mailto:Izadi@ce.sharif.edu)

2 Email: [Movaghar@sharif.edu](mailto:Movaghar@sharif.edu)

1571-0661 © 2009 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2009.08.008

most recently proposed coordination languages, is a channel based exogenous coor- dination language in which complex coordinators are compositionally built out of simpler ones [[1](#_bookmark20),[3](#_bookmark22),[2](#_bookmark21)]. By using Reo specifications, complex component connectors can be organized in a network of channels and build in a compositional manner. Reo relies on a very liberal and simple notion of channels and can model any kind of peer-to-peer communication. The channels used in Reo networks can be considered as simple communicating processes and the only requirements for them are that channels should have two ends (or I/O interfaces), declared to be *sink* or *source* ends, and a user-defined semantics. At source ends data items enter the channel by performing corresponding write operations. Data items are received from a channel at sink ends by performing corresponding read operations. Reo allows for an open ended set of channel types with user defined semantics.

If we want to be able to reason about properties of specifications or verify their correctness, Reo, as well as any other process specification languages, should be given abstract semantics. Generally, when the ultimate aim is reasoning about or verification the properties of specifications, the language is modeled by some kind of transition systems. Labeled transition systems and automata are examples of these semantic models. The key question in giving a semantic model to a specifica- tion language is: ”Whenever can we say that two specifications or two models are equivalent?” Numerous definitions of different equivalence-relations for transition system based models have been presented in the literature of process algebra, au- tomata theory and theories of concurrency. *Trace equivalence* (automata-theoretic equivalence), *weak bisimilarity* presented by Milner [[14](#_bookmark33)] and *failure-based equiva- lences* (CSP-like equivalences) such as the equivalence presented by Hoare [[10](#_bookmark29)] are examples of these equivalences. We say that an equivalence relation *R*1 is *stronger* than *R*2, if whenever two models be equivalent with respect to *R*1, they are also equivalent with respect to *R*2. Equivalence relations *R*1 and *R*2 are *incomparable*, if neither *R*1 is stronger than *R*2, nor *R*2 is stronger than *R*1.

The main goal of verification methods is trying to ascertain that an actual sys- tem or program satisfies its requirements. In formal verification methods one tries to achieve this aim by describing the system using a mathematical model, expressing the requirements as properties of this model and showing through rigorous mathe- matical reasoning that the model of the system indeed has the required properties [[6](#_bookmark25),[13](#_bookmark32)]. If the correctness requirements of a formally modeled computing system are given in a mathematical notion, such as linear or branching time temporal logics or automata on infinite objects, an algorithmic model theoretic process called *model checking* [[6](#_bookmark25)] can be used to check if the system respects its correctness requirements. Model checking has shown to be an efficient and easy to use technique in verifica- tion of computer systems. However, there is a major drawback in using exhaustive model checking: the model of the system tends to be extremely large. In literature this problem is often referred as the *state explosion problem*.

During the last two decades, many methods have been suggested to reduce the number of states which need to be constructed for answering certain verification or analysis questions. Such enhanced state space methods increase substantially the

size of systems that can be verified, while preserving most of the advantages of the model checking method of verification. Compositional verification and its special case, equivalence based compositional reduction [[12](#_bookmark31),[16](#_bookmark35)], partial order reduction by representatives [[15](#_bookmark34)], the pre-order reduction [[9](#_bookmark28)], abstraction [[5](#_bookmark24)] and using the sym- metry properties [[8](#_bookmark27)] are the main of these methods for dealing with state explosion problem. In the compositional verification of a system, one seeks to verify properties of the system from properties of its constituent modules [[6](#_bookmark25),[7](#_bookmark26),[16](#_bookmark35)]. In general, compo- sitional verification may be exploited more effectively when the model is naturally decomposable [[17](#_bookmark36)]. In the method of *equivalence based compositional reduction* components of a system are reduced with respect to an equivalence relation before building the model of the complete system [[9](#_bookmark28),[6](#_bookmark25),[11](#_bookmark30)].

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 also, it should be a *congruence* with respect to the syntactic operators which are used for composing of 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 which 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, introduced by Valmari et al. and called CFFD and NDFD, which have the preservation property: CFFD-equivalence preserves that fragment of linear time temporal logic which has no next-time operator and has an extra operator distinguishing deadlocks [[19](#_bookmark37),[20](#_bookmark38)] and NDFD-equivalence preserves linear time temporal logic without next-time operator [[12](#_bookmark31)]. It was also shown that CFFD and NDFD are the minimal equivalences preserving the above mentioned fragments of linear time temporal logic. In [[20](#_bookmark38)], it was also shown that if we use labeled transition systems as semantic models, CFFD an NDFD are congruences with respect to all composition operators defined in LOTOS.

Constraint automaton, as an extension of finite or Buchi automaton, is a for- malism proposed to capture the operational semantics of Reo [[4](#_bookmark23)]. In a constraint automaton, contrary to finite automata and labeled transition systems, the label of a transition is not a simple character or action name. A transition label con- tains a set of names and a (constraint) proposition. The set of names indicates the names of ports which are participant in doing the transition and the proposition expresses some constraint about the data of the ports. In [[4](#_bookmark23)], constraint automa- ton has been defined as the semantic model of Reo and also, trace-based and weak (bi)simulation-based equivalences of constraint automata have been presented. In this paper, we are interested to investigate failure-based equivalences CFFD and NDFD for constraint automata and their congruency with respect to composition operators which are useful in composing Reo specifications.

The ultimate goal of this paper is to prepare an environment for compositional model checking of Reo specifications modeled by constraint automata using equiva- lence based reduction method. For this purpose we introduce an extended definition of constraint automaton by which every constraint automaton can be considered as

a labeled transition system and each labeled transition system can be translated into a constraint automaton. Also, we introduce two new composition operators for constraint automata: join (production) of two automata with respect to their com- mon port names and hiding of a port name in all transition labels of an automaton. We show that failure-based equivalences CFFD and NDFD are congruence with re- spect to join and hiding operators of constraint automata (see Subsections [4.1](#_bookmark12) and [4.2](#_bookmark16)). Based on these congruency results and because of the linear time temporal logic preservation properties of CFFD and NDFD equivalences and their minimality properties (proved in [[12](#_bookmark31)]), they will useful candidates for compositional reduction of models in the field of model checking in future works.

The paper proceeds as follows: in section 2, we define the notion of *labeled tran- sition systems*, some composition operators for them and two equivalence relations CFFD and NDFD. In section 3, we briefly introduce the traditional definition of constraint automata. Then, we present a new definition of constraint automata by which each labeled transition system can be translated to a constraint automaton and vice-versa. Also, in this section we introduce two composition operators for our defined constraint automata: join (production) and hiding. In section 4, we prove that CFFD and NDFD-equivalences are congruences with respect to join and hiding of constraint automata. In section 5, we conclude and discuss some about the results of this work.

# Preliminaries and Basic Definitions

In this section, we define the notion of *labeled transition systems* and introduce CFFD and NDFD-equivalence relations on them based on the papers introduced them [[19](#_bookmark37),[20](#_bookmark38),[12](#_bookmark31)].

Definition 2.1 A *transition alphabet* is a countable infinite set Σ not containing the empty transition label *τ* . We write Σ*τ* for Σ ∪ {*τ* }, 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 *σ* ∈ (Σ∗ ∪ Σ*ω*), *vis*(*σ*) is used to denote the word obtained by

*τ τ*

removing all *τ* -symbols from *σ* and Σ(*σ*) denote the set of elements of *σ*. A *labeled*

*transition system (lts)* is a triple *L* = (*S, s,* Δ), where *S* is the set of states, *s* ∈ *S* is the initial state and Δ ⊆ *S* × Σ*τ* × *S* is the transition relation. The alphabet of *L* , Σ(*L*) is the following set: Σ(*L*)= {*l* ∈ Σ| ∃*s, s*' : (*s, l, s*') ∈ Δ}. The alphabet of any *lts* is required to be finite.

Now, we recall some basic concepts of process algebra and give the definitions of CFFD and NDFD-equivalences. For a more detailed discussion of these equivalences and the intuitions behind them see [[19](#_bookmark37),[20](#_bookmark38),[12](#_bookmark31)].

Definition 2.2 Let *L* = (*S, s,* Δ) be a labeled transition system (lts).

If *ρ* ∈ Σ∗, we write *s*

*τ*

*ρ*

0 −→

*sn* for *n* = |*ρ*| iff there are *s*1*, ..., s*

*ρ*

*n*−1

such that for all

*ρ*

0 *< i* ≤ *n*, (*si*−1*, ρi, si*) ∈ Δ. If there is an *sn* such that *s*0 −→ *sn* we write *s*0 −→.

If *ρ* ∈ Σ*ω* , we write *s*

*ρ*

iff ∃*s ,s , ...* such that for all *i >* 0, (*s*

*,ρ ,s* ) ∈ Δ. If

*τ* 0 −→ 1 2

*i*−1 *i i*

*σ* ∈ (Σ∗ ∪ *ω*

*σ*⇒ *σ*⇒) iff there is a *ρ* ∈ (Σ∗ ∪ Σ*ω*) such that

Σ ), we write *s*0 = *sn* (*s*0 = *τ τ*

*ρ ρ*

*s*0 −→ *sn*, (*s*0 −→) and *σ* = *vis*(*ρ*).

*σ*

- *σ* ∈ Σ∗

*σ*

is a *trace* of *L* iff *s* =⇒ . *tr*(*L*) is the set of all traces of *L*.

* *σ* ∈ Σ*ω*

is an *inﬁnite trace* iff *s* =⇒ . *inf tr*(*L*) is the set of all infinite traces of *L*.

* *σ* ∈ Σ∗ is a *divergence trace* of *L* iff there is a *ρ* ∈ Σ*ω*

*τ*

*σ* = *vis*(*ρ*). *divtr*(*L*) is the set of all divergence traces of *L*.

*ρ*

such that *s* −→

and

* *s*' ∈ *S* is *stable*, if not *s*' −*τ*→ . Lts *L* is stable if the initial state *s* is stable. We write *stable*(*L*) if *L* is stable, and ¬*stable*(*L*) if it is not.
* (*σ, A*) ∈ Σ∗ × 2Σ, where 2Σ denotes the power set of Σ, is a *failure* of *L* iff there

*σ* '

' *a*

is an *s*' ∈

*S* such that *s* =⇒ *s* and ∀*a* ∈ *A.*¬(*s*

=⇒).

* (*σ, A*) ∈ Σ∗ × 2Σ is a *stable failure* of *L* iff there is a stable *s*' ∈ *S* such that

*σ*

' *a*

*s* =⇒ *s*'

∧ ∀*a* ∈ *A.*¬(*s*

=⇒). *sfail*(*L*) is the set of all stable failures of L.

* (*σ, A*) ∈ Σ∗ × 2Σ is a *nondivergent failure* of *L* iff (*σ, A*) is a failure and *σ* is not a divergence trace. *ndf ail*(*L*) is the set of all nondivergent failures of *L*.
* (*σ, A*) ∈ Σ∗ × 2Σ is a *divergence-masked failure* of *L* iff (*σ, A*) is a failure or *σ* is a divergence trace. *df ail*(*L*) is the set of divergence-masked failures of *L*.
* *σ* ∈ Σ∗ is a *deadlock trace* iff (*σ,* Σ) is a stable failure of *L*. *dtr*(*L*) is the set of all deadlock traces of *L*.
* *σ* ∈ Σ∗ is a *nondivergent deadlock trace* of *L* iff (*σ,* Σ) is a nondivergent failure of *L*. *nddtr*(*L*) is the set of all nondivergent deadlock traces of *L*.Note that, *nddtr*(*L*)= *dtr*(*L*) − *divtr*(*L*).
* The set of all *nondivergent traces* of *L* is *ndtr*(*L*)= {*σ*|(*σ,* ∅) ∈ *ndf ail*(*L*)}.

The following proposition lists some consequences of the definitions for later use (for proofs see [[20](#_bookmark38)]).

Proposition 2.3 *Let L be an lts.*

1. *tr*(*L*)= *divtr*(*L*) ∪ {*σ*|(*σ,* ∅) ∈ *sfail*(*L*)} = *divtr*(*L*) ∪ {*σ*|(*σ,* ∅) ∈ *ndf ail*(*L*)}*.*
2. *tr*(*L*)= {*σ*|(*σ,* ∅) ∈ *f ail*(*L*)} = {*σ*|(*σ,* ∅) ∈ *df ail*(*L*)}*.*
3. *ndf ail*(*L*)= *sfail*(*L*) − (*divtr*(*L*) × 2Σ)*.*
4. *df ail*(*L*)= *sfail*(*L*) ∪ (*divtr*(*L*) × 2Σ)*.*
5. *tr*(*L*)= *divtr*(*L*) ∪ *ndtr*(*L*)*.*
6. *divtr*(*L*) ∩ *ndtr*(*L*)= ∅*.*
7. *If L be a ﬁnite lts, then,*

*inf tr*(*L*)= {*ω* ∈ Σ*ω*|∀*σ* ∈ Σ∗ : (*σ is a proper prefix of ω* → *σ* ∈ *tr*(*L*))}*.*

On the basis of the above definitions and propositions, the equivalence concepts can be easily defined.

Definition 2.4 (i) Let *L* and *L*' be ltss. We say that *L* and *L*' are CFFD equiv- alent and write *L cffd L*' iff *stable*(*L*) ⇔ *stable*(*L*') and *divtr*(*L*) = *divtr*(*L*') and *inf tr*(*L*)= *inf tr*(*L*') and *sfail*(*L*)= *sfail*(*L*').

≈

(ii) Let *L* and *L*' be ltss. We say that *L* and *L*' are NDFD equivalent and write *L ndf d L*' iff *stable*(*L*) ⇔ *stable*(*L*') and *divtr*(*L*) = *divtr*(*L*') and *inf tr*(*L*) = *inf tr*(*L*') and *ndf ail*(*L*)= *ndf ail*(*L*').

≈

The NDFD-equivalence is strictly weaker than CFFD-equivalence in the sense

of the following theorem [[20](#_bookmark38)]:

Proposition 2.5 *If L cffd L*'*, then L ndf d L*'*.*

≈

≈

If the labeled transition systems examined are finite, the component *inf tr* in the above definition is superfluous. This corresponds to the original definition of CFFD- equivalence in [[19](#_bookmark37)], where only finite ltss were considered. Also, it can be shown that it is possible to replace *ndf ail*(*L*) by *df ail*(*L*) in the definition of NDFD-equivalence [[20](#_bookmark38)]. Thus, we have:

Proposition 2.6 *Let L and L*' *be two ﬁnite ltss.*

*cffd*

1. *L* ' ' '

≈ *L iff stable*(*L*) ⇔ *stable*(*L* )*, divtr*(*L*) = *divtr*(*L* )*, and sfail*(*L*) =

*sfail*(*L*')*.*

*ndf d*

1. *L* ' ' '

≈ *L iff stable*(*L*) ⇔ *stable*(*L* )*, divtr*(*L*) = *divtr*(*L* )*, and df ail*(*L*) =

*df ail*(*L*')*.*

If two systems or processes *A* and *B* are CFFD equivalent, then the intuitive meaning is that both have the same computation sequences and furthermore, *A* can deadlock after a sequence of actions if and only if *B* can also deadlock after the same sequence of actions. Also, the computation sequences that lead to a divergence (infinite sequence of internal actions) are the same for the two processes.

Definition 2.7 An equivalence ≈ between ltss is a *congruence* with respect to a

syntactic operator *f* iff for every *L*1*,... , Ln* and *L*' *,... , L*' such that *Li* ≈ *L*' the

1 *n* *i*

following holds: *f* (*L*1*,... , Ln*) ≈ *f* (*L*' *,... , L*' ).

1 *n*

In [[20](#_bookmark38),[12](#_bookmark31)] it has been shown that, CFFD and NDFD equivalences are congruences with respect to the composition operators defined for labeled transition systems based semantics of basic LOTOS and CSP. In the rest of this paper we investi- gate the congruency of these equivalences with respect to composition operators of constraint automata.

# Constraint Automata

Constraint automata have been introduced as the operational semantics of Reo specifications [[4](#_bookmark23)]. In this section, after a short review on Reo, we present the orig- inal definition of constraint automata (as the acceptors of languages of timed data streams, presented in [[4](#_bookmark23)]. Then we introduce an extended definition of them such that they can be considered as labeled transition systems and each label transi- tion system can be translated into a constraint automaton. Also, in this section we introduce two composition operators for our defined constraint automata: join (production) of two constraint automata with respect to their common port names and hiding of a name from all transition labels of a constraint automaton. For more about Reo specification language and constraint automata as its semantics model, see [[1](#_bookmark20),[3](#_bookmark22),[2](#_bookmark21),[4](#_bookmark23)]

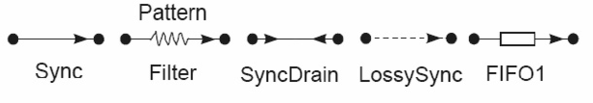


Fig. 1. Basic channel-types in Reo

* 1. *Reo and its Operational Semantics*

Reo is an exogenous coordination language which is based on a calculus of channels [[1](#_bookmark20),[2](#_bookmark21),[4](#_bookmark23)]. By using Reo specifications, complex component connectors can be orga- nized in a network of channels and build in a compositional manner. Reo relies on a very liberal and simple notion of channels and can model any kind of peer-to-peer communication. The only requirements for the channels used in Reo networks are that channels should have two channel ends, declared to be *sink* or *source* ends, and a user-defined semantics. At source ends data items enter the channel by per- forming corresponding write operations. Data items are received from a channel at its sink end by performing corresponding read operations. It has been proved that the set of channel-types shown in Figure [1](#_bookmark5) by their graphical representations, is an expressively complete set of channels [[1](#_bookmark20)]. A complex connector has a graphical representation, called *Reo circuit or network*. 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 *A* are source ends (then *A* is a source node), sink ends (then *A* is a sink node) or whether *A* combines sink and source ends (then *A* is a mixed node). Source and sink nodes represent input and output ports where components might connect to the network. The mixed nodes serves as routers where data items can be transmitted through the network. For more about Reo networks and their examples see [[1](#_bookmark20),[2](#_bookmark21),[3](#_bookmark22),[4](#_bookmark23)].

Now, we can introduce the original definition of constraint automata as the semantic model of Reo [[4](#_bookmark23)].

Definition 3.1 Let *N* be a set of port names and *Data* be a set of data. A *data constraint g* over names set *N* and data set *Data* is a proposition, which can be constructed by using the following grammar:

*g* ::= *true* | *dA* = *d* | *g*1 ∨ *g*2 | ¬*g d* ∈ *Data, A* ∈ *N*

We use *DC*(*N, Data*) as the set of all data constraints over names set *N* and data set *Data*, defined by the above grammar.

Definition 3.2 A *constraint automaton* is a quadruple *C* = (*Q, Names, T, Q*0), where, *Q* is a finite set of states, *Names* is a finite set of names, *Q*0 ⊆ *Q* is the set of initial states, *T* ⊆ *Q* × 2*Names* × *DC*(*Names, Data*) × *Q* is a set of transitions of *C*. If (*p, N, g, q*) ∈ *T* , we call *N* the name set and *g* the guard of the transition. It is required that, *N* /= ∅ and *g* ∈ *DC*(*N, Data*). The constraint automaton *C* is finite iff the sets *Q*, *T* and *Data* be finite.

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

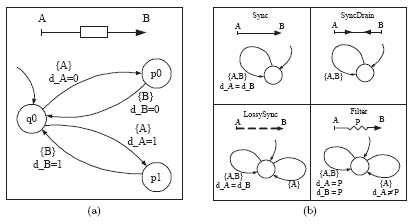


Fig. 2. Constraint automata for some basic channels in Reo

items occur at some of its ports *A*1*, ..., An*. 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*, ..., An*. 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 '→ *d*1 and *A*2 '→ *d*2 resulting in state *p*. If there is no {*A*1*, A*2}-transition from *q* whose data Constraint is fulfilled then *C* rejects. Figure [2](#_bookmark7) shows Constraint automata for some basic channels in Reo.

Definition [3.2](#_bookmark6) is the original definition of constraint automaton (presented in [[4](#_bookmark23)]). In this paper, we will use a modified definition of constraint automaton (such as it will be defined in Definition [3.3](#_bookmark8)). Thus, we sometimes will refer to automaton defined in Definition [3.2](#_bookmark6) as *traditional* constraint automaton.

* 1. *Constraint Automata as Labeled Transition Systems*

Constraint automata, as were presented in [[4](#_bookmark23)] are used as the models of compo- nent connectors of a component based system. A component based system contains both components and connectors. If we consider the whole system, we need to model and compose both components and connectors. Basically, the components can be modeled by labeled transition systems. Thus, we need to find a way to compose Constraint automata and labeled transition systems with each other. For this purpose we can generalize the definition of constraint automaton such that one can consider it as a labeled transition system and also translate labeled transition systems to constraint automata. In this section, we investigate a bidirectional trans- lation between constraint automata and labeled transition systems. We introduce a modified definition of constraint automaton where, transitions can be labeled by internal or external actions. The external actions are as defined in traditional Def- inition [3.2](#_bookmark6). The internal actions are introduced by *τ* labels on transitions. Using this definition, each constraint automaton can be considered as a labeled transition system and each labeled transition system can simply be translated to a constraint automaton. Also, in this section we introduce two composition operators for con- straint automata using their new definition: production (join) of two constraint automata with respect to their common port names and hiding of a port name from

a constraint automaton. We prove that CFFD and NDFD equivalences, introduced in Section [2](#_bookmark1), are congruence with respect to these composition operators.

Definition 3.3 Let *Data* be a set of data. A *constraint automaton* is a quadruple *C* = (*Q, Nam, T, q*0) where, *Q* is a finite set of states, *Nam* is a finite set of names, *T* ⊆ *Q* × (2*Nam* × *DC*(*Nam, Data*)) × *Q* is the transition relation, and *q*0 ∈ *Q* is the initial state.

*N,g*

We write *p* −→ *q* instead of (*p, N, g, q*) ∈ *T* and call *N* the name set and *g* the guard

of the transition. For each (*p, N, g, q*) ∈ *T* , it is required that *g* ∈ *DC*(*N, Data*). Note that *DC*(∅*, Data*) = {*true, false*}. We use *τ* as a shorthand symbol for the transition label (∅*, true*). In other words, transition *p* −*τ*→ *q* is the same as *p* ∅*,true q*.

−→

The main differences of our definition of constraint automaton and its original definition (defined in [[4](#_bookmark23)]) are: 1- In the new definition, *τ* − *transtion* is permitted, while in its original definition it is not. We need *τ* − *transtion* because of two reasons: first that *τ* − *transtion* can be used as a symbol for each kind of internal action which is occurred in the actual system but its real type is not important in modeling by a constraint automaton, second that, the *hiding* operators can hide all port-names of a transition. In such cases, we replace the transition label by *τ* . 2- We supposed that the initial state of a constraint automaton is unique because *τ* − *transtions* are permitted. We can simulate multiple initial states by using a unique initial state and *τ* −*transtions* from it to other possible initial states. 3- Our definition of constraint automaton is departed from the original one by dropping the requirement that all runs have to be infinite. We also deal with finite runs, which are necessary to argue about deadlock configurations.

Obviously, if we consider constraint automata as defined in Definition [3.3](#_bookmark8), each labeled transition system can be translated to a constraint automaton. For this goal, we should save the internal actions by using *τ* − *transitions* and for external actions we should determine their (input or output) ports names and constraints (if they are necessary). On the other hand, each constraint automaton can be considered as a labeled transition system with alphabet Σ = {(*N, g*)|*N* ⊆ *Nam*∧*g* ∈ *DC*(*N, Data*) ∧ *N* /= ∅}. Thus,

Proposition 3.4 *Let C* = (*Q, Nam, T, q*0) *be a constraint automaton over data set* *data, such as deﬁned in Deﬁnition* [*3.3*](#_bookmark8)*. C can be considered as a lts L* = (*S, s,* Δ) *over alphabet* Σ*, such as deﬁned in Deﬁnition* [*2.1*](#_bookmark2)*, where, S* = *Q, s* = *q*0*,*

Σ= {(*N, g*)|*N* ⊆ *Nam* ∧ *g* ∈ *DC*(*N, Data*) ∧ *N* /= ∅}*,*

(*qi,* (*N, g*)*, qj* ) ∈ Δ ⇔ (*qi, N, g, qj* ) ∈ *T and* (*qi, τ, qj*) ∈ Δ ⇔ (*qi, τ, qj*) ∈ *T .*

Based on Proposition [3.4](#_bookmark9), the definitions of traces and failures for constraint automata will be the same as defined in Definition [2.2](#_bookmark3).

* 1. *Composing Constraint Automata*

Because constraint automata are intended to capture the operational semantics of Reo, we need two composition operators for composing or reconstructing them: pro- duction (join) of two constraint automata with respect to their common port names

and hiding a port name from a constraint automaton. In this section, we present our definitions of these two composition operators using the new definition of constraint automaton. These definitions are generalizations of their original counterparts and save all properties proved for them in [[4](#_bookmark23)].

Definition 3.5 Let *C*1 = (*Q*1*,Nam*1*, T*1*, q*01) and *C*2 = (*Q*2*,Nam*2*, T*2*, q*02) be two constraint automata. The *product (join) constraint automaton* of *C*1 and *C*2 is: *C*1 *da C*2 = (*Q*1 × *Q*2*,Nam*1 ∪ *Nam*2*,T, q*01 × *q*02) in which, *T* is defined as follow:

1. If (*q*1*, N*1*, g*1*, p*1) ∈ *T*1, (*q*2*, N*2*, g*2*, p*2) ∈ *T*2, *N*1 /= ∅, *N*2 /= ∅ and *N*1 ∩ *Nam*2 =

*N*2 ∩ *Nam*1, then, (*< q*1*, q*2 *>, N*1 ∪ *N*2*, g*1 ∧ *g*2*,< p*1*, p*2 *>*) ∈ *T* ,

1. If (*q, N, g, p*) ∈ *T*1 and *N* ∩ *Nam*2 = ∅, then, (*< q, q*' *>, N, g, < p, q*' *>*) ∈ *T* ,
2. If (*q, N, g, p*) ∈ *T*2 and *N* ∩ *Nam*1 = ∅, then, (*< q*'*,q >, N, g,< q*'*,p >*) ∈ *T* .

Because the above definition of product is a generalization of its original coun- terpart, if we restrict the alphabet of the language of a constraint automaton to its observable elements (consider all transition labels of the form (*N, g*) as observable and ignore *τ* in all words), it saves all properties which have been shown for its counterpart in [[4](#_bookmark23)].

Definition 3.6 Let *C* = (*Q, Nam, T, q*0) be a constraint automaton and *B* be a name, *B* ∈ *Nam*. The constraint automaton resulted by *hiding* of *B* in *A* is

∃*B*[*C*]= (*Q, Nam*\{*B*}*, T*∃*B , q*0) in which, *T* is defined as follow:

1. If (*q, N, g, p*) ∈ *T* and *N* /= ∅ then (*q, N* \{*B*}*,* ∃*B*[*g*]*, p*) ∈ *T*∃*B*, where

∃*B*[*g*]= ∨*d*∈*Datag*[*dB/d*].

1. If (*q, τ, p*) ∈ *T* then, (*q, τ, p*) ∈ *T*∃*B*.

In addition to the above two composition operators, based on Proposition [3.4](#_bookmark9), we can compose constraint automata using any other well defined composition op- erators defined for label transition systems.

# Congruency Results

In this section we investigate the congruency of CFFD and NDFD equivalences with respect to the join (production) and hiding composition of constraint automata. This section contains two parts, in the first we consider the join operator (as defined in Definition [3.5](#_bookmark10)) and in the other we consider the hiding operator (as defined in Definition [3.6](#_bookmark11)).

* 1. *CFFD and NDFD are congruences with respect to join of constraint automata*

In this section we prove that equivalences CFFD and NDFD are congruences with re- spect to production (join) of finite constraint automata as defined in definition [3.5](#_bookmark10). Our method for the proof is very similar to the methods used by the authors of CFFD and NDFD equivalences for proving their congruences with respect to the composition operators defined in [[20](#_bookmark38)]. First, We define a predicate Join(*σ*; *π, ρ*), which intuitively means that words *π* and *ρ* can be considered as traces of two con- straint automata and word *σ* as a trace in the product (join) constraint automaton

such that *σ* is the result of the production of *ρ* and *π*. Then, we show that the sets of all traces, all stable failures, all divergent traces and all divergence-masked failures of the product automaton can be characterized by their counterparts in the two constraint automaton (see Proposition [4.2](#_bookmark14)). Based on these characterizations, we will prove the congruences. Because our ultimate goal is using of the equivalences in the context of model checking, we will prove the congruences for finite constraint automata.

Definition 4.1 Let *Data* be a finite set of data and *Nam*1 and *Nam*2 be two finite sets of names. Let Σ1 = {(*N, g*)|*N* ⊆ *Nam*1 ∧ *N* /= ∅∧ *g* ∈ *DC*(*N, Data*)}, Σ2 = {(*N, g*)|*N* ⊆ *Nam*2 ∧ *N* /= ∅∧ *g* ∈ *DC*(*N, Data*)},

Σ= {(*N, g*)|*N* ⊆ *Nam*1 ∪ *Nam*2 ∧ *N* /= ∅∧ *g* ∈ *DC*(*N, Data*)} and

*σ* = (*N*1*, g*1)(*N*2*, g*2) *...* be a word over alphabet Σ. The predicate Join(*σ*; *π, ρ*) holds (is true) if and only if the following procedure can obtain words *π* and *ρ* from *σ*, successfully:

* + 1. Define a function *moved* from {1*,* 2*,* 3*,.. .*} to {”*f irst*”*,* ”*second*”*,* ”*both*”} such that: *moved*(*i*)= ”*f irst*” iff *Ni* ∩ *Nam*2 = ∅ and *gi* ∈ *DC*1, *moved*(*i*)= ”*second*” iff *Ni* ∩ *Nam*1 = ∅ and *gi* ∈ *DC*2, otherwise *moved*(*i*)= ”*both*”.
    2. For obtaining *π* from *σ* do:

2-1- for all *i* ≥ 1 which, *moved*(*i*)= ”*both*”, change (*Ni, gi*) to (*Ni* ∩ *Nam*1*, gi*[*Nam*1]),

2-2- remove all (*Ni, gi*) which *moved*(*i*)= ”*second*” from *σ*. 3- For obtaining *ρ* from *σ* do:

3-1- for all *i* ≥ 1 which, *moved*(*i*)= ”*both*”, change (*Ni, gi*) to (*Ni* ∩ *Nam*2*, gi*[*Nam*2]),

3-2- remove all (*Ni, gi*) which *moved*(*i*)= ”*f irst*” from *σ*.

By *g*[*Nami*] we mean the restriction of proposition *g* to the name set *Nami*: it can be obtained by removing all terms containing *dA* = *d* where *A* /∈ *Nami* from the conjunctive normal form of *g*. Obviously, if the above procedure can obtain words *π* and *ρ* successfully, *π* will be a word over alphabet Σ1 and *ρ* will be a word over alphabet Σ2.

Proposition 4.2 *Let C*1 = (*Q*1*,Nam*1*, T*1*, q*01) *and C*2 = (*Q*2*,Nam*2*, T*2*, q*02) *be two ﬁnite constraint automata. Let C* = *C*1 *da C*2 *then,*

* + - 1. *tr*(*C*)= {*σ* | ∃*π* ∈ *tr*(*C*1)*,* ∃*ρ* ∈ *tr*(*C*2)*, Join*(*σ*; *π, ρ*)}*.*
      2. *sfail*(*C*)= {(*σ, A*) | ∃(*π, B*) ∈ *sfail*(*C*1)*,* ∃(*ρ, D*) ∈ *sfail*(*C*2)*, Join*(*σ*; *π, ρ*) ∧

*A* ∩ *G* ⊆ *B* ∩ *D* ∧ *A* ∩ *G*' ⊆ *B* ∪ *D*}*, where,*

*G* = {(*N, g*)| *N* ⊆ *Nam*1 ∪ *Nam*2 ∧ *N* /= ∅∧ (*N* ∩ *Nam*1 = ∅∨ *N* ∩ *Nam*2 = ∅)}*,*

*G*' = {(*N, g*)| *N* ⊆ *Nam*1 ∪ *Nam*2 ∧ *N* /= ∅∧ (*N* ∩ *Nam*1 /= ∅∧ *N* ∩ *Nam*2 /= ∅)}*.*

* + - 1. *stable*(*C*)= *stable*(*C*1) ∧ *stable*(*C*2)*.*

1. *divtr*(*C*)= {*σ* | ∃*π* ∈ *tr*(*C*1)*,* ∃*ρ* ∈ *tr*(*C*2)*,Join*(*σ*; *π, ρ*) *and* (*π* ∈ *divtr*(*C*1) ∨

*ρ* ∈ *divtr*(*C*2))}*.*

1. *df ail*(*C*) = {(*σ, A*) | ∃(*π, B*) ∈ *df ail*(*C*1)*,* ∃(*ρ, D*) ∈ *df ail*(*C*2)*,Join*(*σ*; *π, ρ*) ∧

*A* ∩ *G* ⊆ *B* ∩ *D* ∧ *A* ∩ *G*' ⊆ *B* ∪ *D*} ∪ (*divtr*(*C*1 *da C*2) × 2Σ)*, where,* Σ *is the*

*same as deﬁned in Deﬁnition* [*4.1*](#_bookmark13) *and G and G*' *are the same as deﬁned in (ii).*

Proof. See Appendix [A](#_bookmark39)

Proposition 4.3 *Let C and C*' *be*

*ﬁnite*

*constraint automata over the same set*

*of names, D and D*' *be*

*ﬁnite*

*constraint automata over the same set of names,*

*cffd*

*C* '

*cffd* '

*cffd* ' '

≈ *C and D*

≈ *D . Then, C da D*

≈ *C da D .*

Proof. (i) Based on Proposition [4.2](#_bookmark14)(iii), *stable*(*C da D*) = *stable*(*C*) ∧ *stable*(*D*).

≈

*cffd*

Because of *C* ≈

*C*' and *D cffd*

*D*', we have, *stable*(*C*) = *stable*(*C*') and

*stable*(*D*)= *stable*(*D*'). Thus, *stable*(*C da D*)= *stable*(*C*' *da D*').

1. Based on Proposition [4.2](#_bookmark14)(ii),

*sfail*(*C da D*) = {(*σ, A*) | ∃(*π, B*) ∈ *sfail*(*C*)*,* ∃(*ρ, E*) ∈ *sfail*(*D*)*,Join*(*σ*; *π, ρ*) ∧

*A* ∩ *G* ⊆ *B* ∩ *E* ∧ *A* ∩ *G*' ⊆ *B* ∪ *E*} where,

*G* = {(*N, g*)|*N* ∩ *NamC* = ∅∨ *N* ∩ *NamD* = ∅} and

*G*' = {(*N, g*)|*N* ⊆ *NamC* ∪ *NamD* ∧ *N* /= ∅∧ *N* ∩ *NamC* /= ∅∧ *N* ∩ *NamD* /= ∅}. Because of the CFFD-equivalences *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 da D* are equal

with *G* and *G*' in the case of *C*' *da sfail*(*C*' *da D*').

1. Based on Proposition [4.2](#_bookmark14)(ix),

*D*', respectively. Thus, *sfail*(*C da D*) =

*divtr*(*C da D*)= {*σ* | ∃*π* ∈ *tr*(*C*)*,* ∃*ρ* ∈ *tr*(*D*)*, Join*(*σ*; *π, ρ*)

*and*(*π* ∈ *divtr*(*C*) ∨ *ρ* ∈ *divtr*(*D*))}. Based on Proposition [2.3](#_bookmark4)(a), *tr*(*C*) = *divtr*(*C*) ∪ {*σ*|(*σ,* ∅) ∈ *sfail*(*C*)} and this fact holds also for *C*', *D* and *D*'. Because of CFFD-equivalences *divtr*(*C*) = *divtr*(*C*'), *divtr*(*D*) = *divtr*(*D*'), *sfail*(*C*) = *sfail*(*C*'), *sfail*(*D*)= *sfail*(*D*'). Thus, *tr*(*C*)= *tr*(*C*') and *tr*(*D*)= *tr*(*D*'). Thus, *divtr*(*C da D*)= *divtr*(*C*' *da D*').

Corollary 4.4 *CFFD-equivalence is a congruence with respect to the product (join) of ﬁnite constraint automata.*

Proposition 4.5 *Let C and C*' *be*

*ﬁnite*

*constraint automata over the same set*

*of names, D and D*' *be*

*ﬁnite*

*constraint automata over the same set of names,*

*ndf d*

*C* '

*ndf d* '

*ndf d* ' '

≈ *C and D*

≈ *D . Then, C da D*

≈ *C da D .*

Proof.

The proofs for claims *stable*(*C da D*) = *stable*(*C*' *da*

*D*') and *divtr*(*C da D*) =

*divtr*(*C*' *da D*') are the same as in the proof of Proposition [4.3](#_bookmark15). Now we prove that, *df ail*(*C da D*)= *df ail*(*C*' *da D*').

By Proposition [4.2](#_bookmark14)(x), *df ail*(*C da D*) = {(*σ, A*) | ∃(*π, B*) ∈ *df ail*(*C*)*,* ∃(*ρ, E*) ∈

*df ail*(*D*)*, Join*(*σ*; *π, ρ*) *and A* ∩ *G* ⊆ *B* ∩ *E* ∧ *A* ∩ *G*' ⊆ *B* ∪ *E*} ∪ (*divtr*(*C*1 *da*

*C* ) × 2Σ). Because of *C ndf d C*' and *D ndf d D*', *df ail*(*C*)= *df ail*(*C*'), *df ail*(*D*) =

2 ≈ ≈

*df ail*(*D*') and *divtr*(*C da D*) = *divtr*(*C*' *da D*'). Because of the equality of the

names sets, *G* and *G*' in the case of *C da D* are equal with *G* and *G*' in the case of

*C*' *da D*', respectively. Thus, *df ail*(*C da D*)= *df ail*(*C*' *da D*').

Corollary 4.6 *NDFD-equivalence is a congruence with respect to the product (join) of ﬁnite constraint automata.*

* 1. *CFFD and NDFD are congruences with respect to hiding of constraint au- tomata*

In this section we prove that equivalences CFFD and NDFD are congruences with respect to hiding of port names in finite constraint automata as defined in defini- tion [3.6](#_bookmark11). First, we show that the sets of all traces, all stable failures, all divergent traces and all divergence-masked failures of the automaton after hiding of a port name in a constraint automaton can be characterized by their counterparts in the original constraint automaton (see Proposition [4.8](#_bookmark18)). Based on these characteriza- tions, we will prove the congruences. Because our ultimate goal is using of the equivalences in the context of model checking, we will prove the congruences for finite constraint automata.

Definition 4.7 Let *Nam* be a set of names, *Data* be a set of data, Σ = {(*N, g*)|*N* ⊆ *Nam* ∧ *g* ∈ *DC*(*N, Data*)} and *B* ∈ *Nam*. We define the set hide *B* in Σ1, for each set Σ1 ⊆ Σ and the word hide *B* in *σ*, for each word *σ* = (*N*1*, g*1)(*N*2*, g*2) *...* such that:

hide *B* in Σ1 = {(*N* \{*B*}*,* ∃[*B*]*g*)|(*N, g*) ∈ Σ1 ∧ *N* /= {*B*}∧ *N* /= ∅}.

hide *B* in *σ* is the word that is obtained after removing all pairs of the form (∅*, g*) from word (*N*1\{*B*}*,* ∃[*B*]*g*1)(*N*2\{*B*}*,* ∃[*B*]*g*2) *.. .*.

Proposition 4.8 *Let C* = (*Q, Nam, T, q*0) *be a ﬁnite constraint automaton and A* = ∃*B*[*C*] *be the constraint automaton resulted by hiding of B in C (for B* ∈ *Nam). Then,*

1. *tr*(*A*)= {*hide B in σ*| *σ* ∈ *tr*(*C*)}*.*
2. *sfail*(*A*)= {(*hide B in σ, A*)|(*σ, A* ∪ *A*' ∪ ^ ∈ *sfail*(*C*)}}*, where*

*B*)

*A*' = {(*N* ∪ {*B*}*, g*)| ∃*g*' ∈ *DC*(*N, data*): (*N, g*') ∈ *A*} *and*

^

*B* = {({*B*}*, g*)| *g* ∈ *DC*({*B*}*, data*)}*.*

1. *stable*(*A*)= *stable*(*C*) ∧ ∀*g* ∈ *DC*({*B*}*, Data*): ({*B*}*, g*) /∈ *tr*(*C*)*.*
2. *divtr*(*A*) = {*hide B in σ*| *σ* ∈ *divtr*(*C*)} ∪ {*hide B in σ*|*σ* ∈ *inf tr*(*C*) ∧

|*hide B in σ*| *<* ∞}*.*

1. *df ail*(*A*)= {(*hide B in σ, A*)| (*σ, A*∪*A*'∪ ^ ∈ *df ail*(*C*)} ∪ (*divtr*(∃*B*[*C*])×2Σ)*,*

*B*)

*where,* Σ *is so deﬁned in Deﬁnition* [*4.7*](#_bookmark17)*.*

Proof. See Appendix [B](#_bookmark40)

Proposition 4.9 *Let C and C*' *be ﬁnite constraint automata over the same set of*

*cffd*

'

*names, C*

*cffd* '

≈ *C and B be a name in the set of names. Then,* ∃*B*[*C*]

Proof.

≈ ∃*B*[*C* ]*.*

1. By Proposition [4.8](#_bookmark18)(iii), *stable*(∃*B*[*C*]) = *stable*(*C*) ∧ ∀*g* ∈ *DC*({*B*}*, Data*) :

({*B*}*, g*) /∈ *tr*(*C*). Because, *C cffd C*', *stable*(*C*)= *stable*(*C*'), *divtr*(*C*)= *divtr*(*C*') and *sfail*(*C*) = *sfail*(*C*'). By Proposition [2.3](#_bookmark4)(a), *tr*(*C*) = *divtr*(*C*) ∪ {(*σ,* ∅)|*σ* ∈ *sfail*(*C*)}. Thus, *tr*(*C*)= *tr*(*C*'). Therefore, *stable*(∃*B*[*C*]) = *stable*(∃*B*[*C*']).

≈

1. By Proposition [4.8](#_bookmark18)(ii),

*sfail*(∃*B*[*C*]) = {(hide *B* in *σ, A*)| (*σ, A* ∪ *A*' ∪ ^

*B*)

∈ *sfail*(*C*)}}, *A*' = {(*N* ∪

{*B*}*, g*)| ∃*g*' ∈ *DC*(*N, data*) : (*N, g*') ∈ *A*} ^ {({*B*}*, g*)| *g* ∈ *DC*({*B*}*, data*)}.

, *B* =

Because, *C cffd C*', *sfail*(*C*)= *sfail*(*C*'). Because the name sets of *C* and *C*' are

≈

^

equal, the definitions of sets *A*' and *B* in the cases of *C* and *C*' are the same. Thus,

*sfail*(∃*B*[*C*]) = *sfail*(∃*B*[*C*']).

1. By Proposition [4.8](#_bookmark18)(ix), *divtr*(∃*B*[*C*]) = {hide *B* in *σ*| *σ* ∈ *divtr*(*C*)}∪

{hide *B* in *σ*|*σ* ∈ *inf tr*(*C*) ∧ |hide *B* in *σ*| *<* ∞}. As we showed in (i),

*tr*(*C*)= *tr*(*C*'). By Proposition [2.3](#_bookmark4)(g) and because *C* and *C*' are finite,

*inf tr*(*C*)= {*ω* ∈ Σ*ω*|∀*σ* ∈ Σ∗ : (*σ is a proper prefix of ω* → *σ* ∈ *tr*(*C*))}. Thus,

*inf tr*(*C*)= *inf tr*(*C*') and *divtr*(∃*B*[*C*]) = *divtr*(∃*B*[*C*']).

Corollary 4.10 *CFFD-equivalence is a congruence with respect to the hiding of port names from ﬁnite constraint automata.*

Proposition 4.11 *Let C and C*' *be ﬁnite constraint automata over the same set of*

*ndf d*

*names, C* '

*ndf d* '

≈ *C and B be a name in the set of names. Then,* ∃*B*[*C*] ≈ ∃*B*[*C* ]*.*

Proof. The proofs for claims *stable*(∃*B*[*C*]) = *stable*(∃*B*[*C*']) and *divtr*(∃*B*[*C*]) =

*divtr*(∃*B*[*C*']) are the same as in the proof of Proposition [4.9](#_bookmark19). By Proposition [4.8](#_bookmark18)(x),

*df ail*(∃*B*[*C*]) = {(hide *B* in *σ, A*)| (*σ, A* ∪ *A*' ∪ ^ ∈ *df ail*(*C*)} ∪ (*divtr*(∃*B*[*C*]) ×

*B*)

2Σ). Because, *C ndf d C*', *df ail*(*C*) = *df ail*(*C*'). As we showed, *divtr*(∃*B*[*C*]) =

≈

*divtr*(∃*B*[*C*']). Thus, *df ail*(∃*B*[*C*]) = *df ail*(∃*B*[*C*']).

Corollary 4.12 *NDFD-equivalence is a congruence with respect to the hiding of port names from ﬁnite constraint automata.*

# Conclusions

The ultimate goal of this paper was to prepare an environment for compositional model checking of Reo specifications modeled by constraint automata using equiv- alence based reduction method. For this purpose we introduced an extended def- inition of constraint automaton by which every constraint automaton can be con- sidered as a labeled transition system and each labeled transition system can be translated into a constraint automaton. Also, we introduced two new composition operators for constraint automata: join (production) of two automata with respect to their common port names and hiding of a port name in all transition labels of an automaton. We showed that failure-based equivalences CFFD and NDFD are congruence with respect to join and hiding operators of constraint automata.

Based on these congruency results and because of the linear time temporal logic preservation properties of CFFD and NDFD equivalences and their minimality prop- erties (proved in [[12](#_bookmark31)]), they will useful candidates for compositional reduction of models in the field of model checking in future works. For this purpose, we need to have algorithms for reducing sizes of constraint automata with respect to the equivalence relations. Naturally, the algorithms will be generalizations of ordinary

algorithms for converting an automaton into its deterministic counterpart and then minimization of it.

# References

1. Arbab F., *Reo: A Channel-based Coordination Model for Component Composition*, Math. Struc. in Computer Science, 14(3), (2004), 329-366.
2. Arbab F., *Abstract Behaiviour Types: A foundation model for components and their composition*, science of Computer Programming, 55, (2005), 3-52.
3. Arbab F., Mavadat F., *Coordination Through Channel Composition*, Proceedings of Coordination Languages and Models 2002, LNCS, 2315, Springer-Verlag, (2002).
4. Baier C., Sirjani M., Arbab F., Rutten J., *Modelling Component connectors in Reo by Constraint Automata*, Science of Computer Programming, 61, (2006), 75-113.
5. Clarke E., Grumberg O., Long D., *Model Checking and Abstraction*, ACM Transactions on Programming Languages and Systems, 16(5), (1994), 1512-1542.
6. Clarke E., Grumberg O., Peled D., “Model Checking”, The MIT Press, (1999).
7. Clarke E., Long D., McMillan K., *Compositional Model Checking*, Proceeding of the 4th IEEE Symposium on Logic in Computer Science, (1989), 353-362.
8. Emerson A., Sistla A., *Symmetry and Model Checking*, Proceedings of CAV’93, (1993), 463-478.
9. Graf S., Steffen B., *Compositional Minimization of Finite-State Systems*, Proceedings of CAV’90, Springer-Verlag, (1991), 186-196.
10. Hoare C.A.R., ”Communicating Sequential Processes”, Prentice-hall, (1985).
11. 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), Elsevier (2005).
12. 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.
13. Manna Z., Pnueli A., ”The Temporal Logic of Reactive and Concurrent Systems: Specification”, Springer-Verlag, (1991).
14. Milner R., ”Communication and Concurrency”, Prentice-Hall, (1989).
15. Peled D., *Verification for Robust Specification*, Conference on Theorm Proving in Higher Order Logic, Springer-Verlag, (1997), 231-241.
16. 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.
17. 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).
18. Valmari A., *Failur-based Equivalences are Faster than Many Believe*, Proc. Structures in Concurrency Theory, May 1995, Springer-Verlag (1995), 326-340.
19. 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.
20. Valmari A., Tienari M., *Compositional Failure Based Semantic Models for Basic LOTOS*, Formal Aspects of Computing 7, (1995), 440-468.

# Proof of Proposition [4.2](#_bookmark14)

In this section, we present the proofs of all parts of Proposition [4.2](#_bookmark14).

* 1. This proposition is a direct consequence of Definitions [2.2](#_bookmark3), [3.5](#_bookmark10) and [4.1](#_bookmark13).
  2. Let (*π, B*) ∈ *sfail*(*C*1), (*ρ, D*) ∈ *sfail*(*C*2) and *Join*(*σ*; *π, ρ*). We prove that for all *A* ⊆ Σ, if *A* ∩ *G* ⊆ *B* ∩ *D* ∧ *A* ∩ *G*' ⊆ *B* ∪ *D*, then, (*σ, A*) ∈ *sfail*(*C*1 *da C*2). First note that, *π* ∈ *tr*(*C*1), *ρ* ∈ *tr*(*C*2) and *Join*(*σ*; *π, ρ*), thus based on Proposition

[4.2](#_bookmark14)(i), *σ* ∈ *tr*(*C*1 *da 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 *da C*2 after tracing

*σ* (We denote this state by *qF* , the last state in *C*1 after tracing *π* by *qB* and the last state in *C*2 after tracing *ρ* by *qD*). Let *A* be the greatest subset of Σ where *A* ∩ *G* ⊆ *B* ∩ *D* ∧ *A* ∩ *G*' ⊆ *B* ∪ *D* and (in the way of proof by contradiction)

suppose that there is an outgoing transition from state *qF* in *C*1 *da C*2 with label

(*N, g*) ∈ *A*. Based on Definition [3.5](#_bookmark10), *N* should have one of the three following forms:

(1) *N* = *N*1 where *N*1 ⊆ *Nam*1 and *N* ∩ *Nam*2 = ∅. In this case, (*N, g*) ∈ *A* ∩ *G*. Thus, (*N, g*) ∈ *B* ∩ *D*. But, both (*ρ, D*) and (*π, B*) are fails in their corresponding automata. Thus, it is impossible that (*N, g*) be the label of an outgoing transition from *qF* in the product automaton. (2) *N* = *N*2 where *N*2 ⊆ *Nam*2 and *N* ∩*Nam*1 =

∅. The proof is symmetric with case (1). (3) *N* = *N*1 ∪ *N*2 where *N*1 ⊆ *Nam*1, *N*2 ⊆ *Nam*2 and *N*1 ∩ *Nam*2 = *N*2 ∩ *Nam*1. In this case, (*N, g*) ∈ *A* ∩ *G*'. Thus, (*N, g*) ∈ *B* or (*N, g*) ∈ *D*. In both cases it is impossible that that (*N, g*) be the label of an outgoing transition from *qF* in the product automaton, because at least one of states *qB* and *qD* does not have an outgoing transition with label (*N, g*) in their corresponding automaton. Because we supposed that *A* is the greatest subset of Σ where *A* ∩ *G* ⊆ *B* ∩ *D* ∧ *A* ∩ *G*' ⊆ *B* ∪ *D*, our claim holds for the smaller subsets of Σ.

On the other hand, let (*σ, A*) ∈ *sfail*(*C*1 *da*

*C*2). Thus *σ* ∈ *tr*(*C*1 *da*

*C*2) and

based on Proposition [4.2](#_bookmark14)(i), ∃*π* ∈ *tr*(*C*1)*,* ∃*ρ* ∈ *tr*(*C*2)*, Join*(*σ*; *π, ρ*). Let *B* be the greatest subset of Σ where (*π, B*) ∈ *f ail*(*C*1) and *D* be the greatest subset of Σ where (*ρ, D*) ∈ *f ail*(*C*2). Again, we denote the last state in *C*1 after tracing *π* by *qB*, the last state in *C*2 after tracing *ρ* by *qD* and the last state in *C*1 *da C*2 after tracing *σ* by *qF* . Because *qF* is stable and based on Definition [3.5](#_bookmark10), *qB* and *qD* are stable. Thus, (*π, B*) and (*ρ, D*) are stable failures. If (*N, g*) ∈ *A* ∩ *G* then *N* ∩ *Nam*1 = ∅ or *N* ∩ *Nam*2 = ∅ and there is no outgoing transition with label (*N, g*) from *qF* . If *N* ∩ *Nam*1 = ∅ then obviously, (*N, g*) ∈ *B* and based on Definition [3.5](#_bookmark10) it can not be the label of an outgoing transition from *qD* in *C*2 . Thus, because of the maximality of *D*, (*N, g*) ∈ *D*. Thus, (*N, g*) ∈ *B* ∩ *D*. Similarly, if *N* ∩ *Nam*2 = ∅ then (*N, g*) ∈ *B* ∩ *D*. Thus, *A* ∩ *G* ⊆ *B* ∩ *D*. If (*N, g*) ∈ *A* ∩ *G*' then *N* ∩ *Nam*1 /= ∅ and *N* ∩ *Nam*2 /= ∅. Let (in the way of proof by contradiction) that (*N, g*) /∈ *B* ∪ *D*. Thus, there are an outgoing transition with label (*N, g*) from *qB* in *C*1 and an outgoing transition with label (*N, g*) from *qD* in *C*2 and based on Definition [3.5](#_bookmark10)

there is an outgoing transition with label (*N, g*) from *qF* in *C*1 *da*

contradicts that (*σ, A*) is a failure.

*C*2. But this

* 1. , (ix) These propositions are direct consequences of Definitions [2.2](#_bookmark3) and [3.5](#_bookmark10).

(x) By Proposition [2.3](#_bookmark4)(d), *df ail*(*C*1 *da C*2) × 2Σ). By using [4.2](#_bookmark14)(ii),

*C*2) = *sfail*(*C*1 *da*

*C*2) ∪ (*divtr*(*C*1 *da*

*df ail*(*C*1 *da C*2)= {(*σ, A*) | ∃(*π, B*) ∈ *sfail*(*C*1)*,* ∃(*ρ, D*) ∈ *sfail*(*C*2)*,*

*Join*(*σ*; *π, ρ*)∧*A*∩*G* ⊆ *B* ∩*D*∧*A*∩*G*' ⊆ *B* ∪*D*}∪(*divtr*(*C*1 *da C*2)×2Σ)*.* (∗∗) The Equation (\*\*) contains two instance of *sfail* and we need to show that the replacement of both by *df ail* 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 *df ail* adds some pairs to the set {(*σ, A*) |*...*} in the righthand side of the equation,

but all of these new pairs are in (*divtr*(*C*1 *da*

*C*2) × 2Σ). Thus, the union set (the

righthand side of the equation) does not change. For this goal, first suppose that we replace *sfail*(*C*1) by *df ail*(*C*1). Because, *df ail*(*C*1)= *sfail*(*C*1) ∪ (*divtr*(*C*1) × 2Σ) (see Proposition [2.3](#_bookmark4)(d)), the only effect of replacement is that new pairs (*σ, A*) may be introduced related to some (*π, B*) and (*ρ, D*) such that *π* ∈ *divtr*(*C*1), (*ρ, D*) ∈ *sfail*(*C*2) and *Join*(*σ*; *π, ρ*) hold. But then *ρ* ∈ *tr*(*C*2), and by the replacement of *sfail* by *df ail*, (*σ, A*) belongs to (*divtr*(*C*1) × 2Σ). By a symmetric argument, we can show that the replacement of the other *sfail* by *df ail* does not change the righthand side of the Equation (\*\*).

# Proof of Proposition [4.8](#_bookmark18)

In this section, we present the proofs of all parts of Proposition [4.8](#_bookmark18).

* 1. This is a direct consequence of Definitions [2.2](#_bookmark3) and [4.7](#_bookmark17).
  2. If (*ρ, A*) ∈ *sfail*(∃*B*[*C*]), then for the automaton (∃*B*[*C*]), we know that

*a*

there is a state *q* ∈ *Q* which, *q*

0*,B*

*ρ*

=⇒ *q* and *stable*(*q*) and ∀*a* ∈ *A* (¬ *q*

=⇒).

Because *ρ* is a trace in ∃*B*[*C*], there is a trace *σ* ∈ *tr*(*C*) such that *ρ* = hide *B* in *σ*,

*σ*

Σ(*ρ*) = hide *B* in Σ(*σ*) and in automaton *C*, *q*0 =⇒ *q*. Because, *q* is stable in

∃*B*[*C*], there is no transition of the form *q* →*τ B q*', by using the definition of hiding,

there is no transition of the form *q* →*τ*

we prove that (*σ, A* ∪ *A*' ∪ ^

*q*' in *C*. Thus, *q* is also stable in *C*. Now

*B*) is a failure of *C*. First note that because (*ρ, A*) is a

failure of ∃*B*[*C*], for every (*N, g*) ∈ *A*, *B* /∈ *A*. Thus *A* and *A*' are two disjoint sets. Because (*ρ, A*) is a failure in ∃*B*[*C*] and *ρ* = hide *B* in *σ*, (*σ, A*) is a failure of *C*. For set *A*', we know that *A* = hide *B* in *A*'. Thus, (*σ, A*') is also a failure of *C*. Because *q* is stable in ∃*B*[*C*], by the definition of hiding, there is no transition of

{*B*}*,g*

'

the form *q* −→ *q*

^

in *C*. Thus, (*σ, B*) is a failure of *C*. As the overall consequence:

*sfail*(∃*B*[*C*]) ⊆ {(hide *B* in *σ, A*)| (*σ, A* ∪ *A*' ∪ ^ }.

*B*)

On the other hand, let (*σ, A* ∪ *A*' ∪ ^ ∈ *sfail*(*C*)} and *ρ* = hide *B* in *σ*. Thus,

*B*)

*σ*

for the automaton *C*, we know that there is a state *q* ∈ *Q* which, *q*0 =⇒ *q* and

*stable*(*q*) and ∀*a* ∈ *A* ∪ *A*' ∪ ^ ¬ *a*⇒ *σ*⇒ *q* is a run of *C* and

*B,* (

*ρ*

*q* = ). Because, *q*0 =

*ρ* = hide *B* in *σ*, *q*0*,B* =⇒ *q* is a run of ∃*B*[*C*]. Because, in the automaton *C* there

is no transition of the form *q* →*a q*' in which, *a* ∈ *A* ∪ *A*', by using the definition of

hiding, there is no transition of the form *q* →*a q*' in which, *a* ∈ *A* in the automaton

∃*B*[*C*]. Thus, (*ρ, A*) is a failure of ∃*B*[*C*]. Because, *q* is stable in *C* and there is

no transition of the form *q* →*a q*', *a* ∈ {({*B*}*, g*)|*g* ∈ *DC*({*B*}*, data*)}, *q* is stable

in ∃*B*[*C*]. Thus, (*ρ, A*) is a stable failure of ∃*B*[*C*]. As the overall consequence:

{(hide *B* in *σ, A*)| (*σ, A* ∪ *A*' ∪ ^ } ⊆ *sfail*(∃*B*[*C*])}.

*B*)

* 1. (ix) These are direct consequences of Definitions [2.2](#_bookmark3) and [3.6](#_bookmark11).

(x) By Proposition [2.3](#_bookmark4)(d), *df ail*(∃*B*[*C*]) = *sfail*(∃*B*[*C*]) ∪ (*divtr*(∃*B*[*C*]) × 2Σ). Thus, using [4.8](#_bookmark18)(ii),

*df ail*(∃*B*[*C*]) = {(hide *B* in *σ, A*)| (*σ, A* ∪ *A*' ∪ ^ ∈ *sfail*(*C*)}} ∪ (*divtr*(∃*B*[*C*]) ×

*B*)

2Σ) (∗)

The only effect of replacement of *sfail* by *df ail* in Equation (\*) is that new pairs (hide *B* in *σ, A*) may be introduced where, *σ* ∈ *divtr*(*C*). But obviously based on Definition [4.7](#_bookmark17), if *σ* ∈ *divtr*(*C*) then, hide *B* in *σ* ∈ *divtr*(∃*B*[*C*])). Thus, the replacement of *sfail* by *df ail* in Equation (\*) does not change the righthand side of it.