Technical University of Dresden **Faculty of Computer Science**

International Center for Computational Logic

Symmetry Reduction for Reo and Constraint Automata

Master's Thesis

Submitted in partial fulfillment of master's degree in

Computational Logic

by

Amin Timany

(Immatriculation Number: 3736201)

To the International Center for Computational Logic,
Faculty of Computer Science,
Technical University of Dresden

Supervisors:	Dr. Sascha Klüppelholz, Dr. Joachim Klein
Professor in charge:	Prof. Dr. rer. nat. Christel Baier
Department:	Chair of Algebraic and Logic Foundations of Computer Science

Date and Place: *May 28, 2013, Dresden, Germany*

Declaration

I, Amin Timany, the holder of the information below, hereby declare that this thesis, including the text and graphics, are my own personal work. Furthermore, I declare that I have used no materials other than those referenced in the bibliography at the end of this thesis.

First Name:	Amin
Last Name:	Timany
_____	_____
_____	_____
_____	_____

Date and Place: *May 28, 2013, Dresden, Germany*

Acknowledgement

I would like to take this opportunity to thank and show my deepest gratitude towards all those whom without their support, writing this thesis would not have been possible.

I would like to extend my deepest gratitude towards the support of Prof. Baier and my supervisors, Dr. Klüppelholz and Dr. Klein. For giving me the opportunity to work with them and their invaluable directions, discussions, constructive criticism and encouragements.

Moreover, I could not be grateful enough towards my friends and family, especially my kind parents and dear wife, who helped provide a stress free environment for my work and were encouraging and supporting all the way through my studies and in particular, during writing this master thesis.

Last, but definitely not the least, I should thank technical university of Dresden and in particular International Center for Computational Logic (ICCL), for providing me with the chance to study there.

Abstract

Model checking is an automated approach to verification of systems, e.g., software systems, hardware systems, communication protocols, etc. This is done via representing the system as a transition system, that is, representing different situations that the system can be in as a number of *states* and their relations, while representing properties that have to be verified as temporal logic formulas. Unfortunately, when the system at hand is comprised of a number of subsystems (processes or components) that run in parallel, the number of states of the system grows exponentially in the number of sub-systems that run in parallel. For tackling the phenomenon, called *state space explosion problem*, there have been several approaches developed. A well known and widely practiced approach is *symmetry reduction* that makes use of symmetry of sub-systems running in parallel to reduce the number of states of the composite system.

Constraint automata are a variation of transition systems where, there are a number of data-flow locations allowing transmutation of data. The name constraint automata refers to the fact that each transition puts a number of restrictions (constraints) on the data flow throughout constraint automata as that transition is being executed. Reo is an exogenous channel based communication language providing orchestration among components of a system. The semantics underlying Reo circuits can be expressed as constraint automata. In this thesis, we adapt the technique of symmetry reduction to the case of systems expressed in constraint automata and discuss how Reo circuits can be helpful in isolating symmetrical parts of the system, facilitating more efficient dealing with symmetry reduction.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Model Checking and State Space Explosion

Model checking is an automated technique to verify properties of systems, e.g., software systems, hardware systems, communication protocols, etc. In this technique, models are specified as a transition system, while the properties to be checked are expressed in temporal logics. It was introduced by Clarke and Emerson in [CE82] and by Queille and Sifakis in [QS82], independently.

A transition system consists of a set of states S, a relation $R \subseteq S \times S$ on those states called the transition relation and a set of initial states (subset of S). At the beginning the system is assumed to be in an initial state and after passage of each temporal unit the system moves to a new state following one of the transitions originating in that state.

Properties, on the other hand, are specified in temporal logics which specify conditions on states of the system over time; e.g., "the system is never in a deadlock situation", "after a finite time after receiving a request that request is acknowledged" and so on.

In such a scenario a model checker tool, given a model (specified as a transition system) and a property (expressed as a temporal logic formula), either confirms the given property or gives a counter example in the given model.

As an example, consider the example of a transition system depicted in Figure 1.1. It depicts the transition system of a process that is initially in standby state, i.e., 'st'. It then goes to a state where it requests two resources, i.e., '$\mathbf{r_1 r_2}$' where $\mathbf{r_i}$ stands for "request for

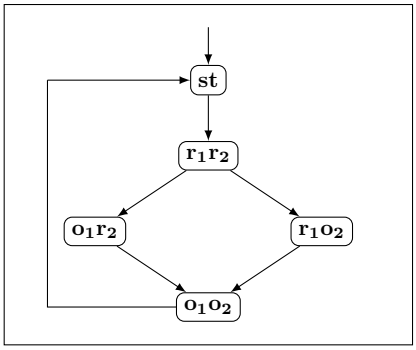


Figure 1.1: An example of a transition system of a process requesting and acquiring two resources.

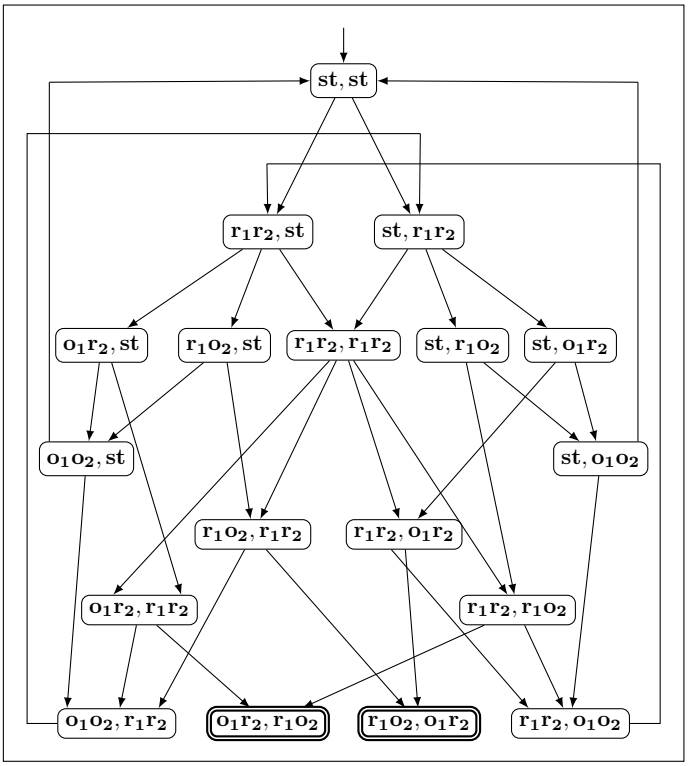


Figure 1.2: The transition system of interleaving two instances of process in Figure 1.1 after restricting resources – a resource cannot be owned by both processes.

resource number i", and after owning them both, i.e., 'o_1o_2' where o_i stands for "resource number i is owned by the process", it goes back to standby state.

Figure 1.2, on the other hand, illustrates a system where two instance of the process depicted in Figure 1.1 are running in parallel. Furthermore, in the system of Figure 1.2, states of the system are restricted to the cases where resource ownerships are not violated, i.e., no resource is owned by more than one process. In this system, though, there has been no measures taken to avoid deadlocks, i.e., situations in which each process owns a resource and is waiting for the other (which is owned by the other process). In Figure 1.2, the states where a deadlock occurs are distinguished by double line borders.

Clearly, in Figure 1.2, the deadlock states are reachable which is undesirable. Therefore, if this system is given to a model checker together with a temporal logic formula specifying "never deadlock", the system will reject the property and provides a counter example of an execution path where a deadlock state is reached.

As it is evident in the example of Figure 1.1 and Figure 1.2, the number of states of a system increases exponentially in the number of processes (or components of the system) that are run in parallel. In this case, the system in Figure 1.1 has 5 states while the system in Figure 1.2 has 18 states (there are 7 states removed due to having both processes owning the same resource). There are a number of approaches that have been introduced and investigated so far among which are techniques like symmetry reduction. We will discuss some of these approaches and in particular symmetry reduction in the next section.

For a rigorous and more precise consideration of the subject, the reader is kindly referred to [CK96, CW96, Wol95, Mer01, BK08].

1.2 Approaches to Tackle State Space Explosion

The state space explosion problem refers to the problem of an exponential increase of the number of states of the composite system in the number processes (or components of the system) that run in parallel. During past decades there have been many approaches developed to tackle this problem. These approaches use different aspects of the nature of this exponential blow up (or the nature of the model checking problem) to tackle this problem. These approaches include but are not limited to: symbolic model checking, model slicing, partial order reduction, symmetry reduction.

Here, we will give an overview of these approaches while focussing on *symmetry reduction* which is the main subject of this thesis. As it is mentioned in [Pel09], the best way for a system to achieve efficiency, the best measure to take is not to be limited to a single approach and rather combine different approaches for an added merit. For further detail about approaches to tackle the state space explosion problem, refer to [Pel09, CGJ+01, BCM+92, McM92, CMCHG96, KB07, EW03, HDZ00, DHH+06, Pel93, HP94, Pel96, GP93, GvLH+96, Val91b, Val91a, KLY02, BP02, CEFJ96, CJ95, GS05, ID96, CEJS98, ES97, Ios02, MDC06, SG04, ET03, DM05, DMP07, DM06, KNP06, BDE+12].

Symbolic Model Checking In this approach, introduced by Burch et. al. in [BCM+92], a number of binary variables (variables with domain $\mathbb{B} = \{0, 1\}$) are considered to represent the transition system. States, the set of initial states and transition relation of the transition system, are in turn represented by binary functions (functions of the type $\mathbb{B}^n \to \mathbb{B}$), over these variables or two copies of them representing before and after states in case of transitions.

Example 1.1. As an example, consider the system in Figure 1.1. This system can be represented by three binary variables v_{st}, v_{r_1} and v_{r_2}. We can simply have the set of states, the set of initial states and transition relation represented as binary functions over v_{st}, v_{r_1} and v_{r_2} as follows:

$$
\begin{aligned}
f_{st}(v_{st}, v_{r_1}, v_{r_2}) &= v_{st} \\
f_{r_1 r_2}(v_{st}, v_{r_1}, v_{r_2}) &= \overline{v_{st}} \wedge \overline{v_{r_1}} \wedge \overline{v_{r_2}} \\
f_{o_1 r_2}(v_{st}, v_{r_1}, v_{r_2}) &= \overline{v_{st}} \wedge v_{r_1} \wedge \overline{v_{r_2}} \\
f_{r_1 o_2}(v_{st}, v_{r_1}, v_{r_2}) &= \overline{v_{st}} \wedge \overline{v_{r_1}} \wedge v_{r_2} \\
f_{o_1 o_2}(v_{st}, v_{r_1}, v_{r_2}) &= \overline{v_{st}} \wedge v_{r_1} \wedge v_{r_2}
\end{aligned}
$$

$$
f_{init}(v_{st}, v_{r_1}, v_{r_2}) = v_{st}
$$

$$
\begin{aligned}
f_{rel}(v_{st}, v_{r_1}, v_{r_2}, v'_{st}, v'_{r_1}, v'_{r_2}) = \quad & (v_{st} \wedge \overline{v'_{st}} \wedge \overline{v'_{r_1}} \wedge \overline{v'_{r_2}}) \\
\vee \quad & (\overline{v_{st}} \wedge \overline{v_{r_1}} \wedge \overline{v_{r_2}} \wedge \overline{v'_{st}} \wedge v'_{r_1} \wedge \overline{v'_{r_2}}) \\
\vee \quad & (\overline{v_{st}} \wedge \overline{v_{r_1}} \wedge \overline{v_{r_2}} \wedge \overline{v'_{st}} \wedge \overline{v'_{r_1}} \wedge v'_{r_2}) \\
\vee \quad & (\overline{v_{st}} \wedge v_{r_1} \wedge \overline{v_{r_2}} \wedge \overline{v'_{st}} \wedge v'_{r_1} \wedge v'_{r_2}) \\
\vee \quad & (\overline{v_{st}} \wedge \overline{v_{r_1}} \wedge v_{r_2} \wedge \overline{v'_{st}} \wedge v'_{r_1} \wedge v'_{r_2}) \\
\vee \quad & (\overline{v_{st}} \wedge v_{r_1} \wedge v_{r_2} \wedge v'_{st})
\end{aligned}
$$

where, the line above variable names represents negation and $\vee$ and $\wedge$ represent disjunction and conjunction respectively; while, f_{st}, $f_{r_1r_2}$, $f_{o_1r_2}$, f_{r_1,o_2} and $f_{o_1o_2}$ each represent the binary functions of their subscript, f_{init} is the binary function representing the set of initial states and f_{rel} is the binary function representing the transition relation of the system. ■

Using binary representations helps compress the size of transition systems while giving the opportunity of using very efficient algorithms (see [Bry86, Ake78]) for the manipulation of sets of states which in turn helps alleviate the state space explosion problem. For further details on symbolic model checking see [McM92, BCM+92, CMCHG96, KB07, EW03].

Model Slicing Slicing is a technique introduced by Mark Weiser in [Wei84] for computing portions of a program that are relevant to a particular part of the program which is of interest for analysis. In this technique, given a slicing criterion, i.e., a line of the program and some variables whose value is important at that line, the program code is analyzed to produce a fragment of the system that (potentially) has an effect on the value of the variables of the given slicing criterion at the line specified in the given slicing criterion. For further reading on the subject of program slicing, the reader can refer to [Wei84, Tip95].

This approach can be and has been lifted to the case of model checking. This approach is most useful where the model is expressed in a form of guarded command language (see [Dij78]) or is generated from the source code of a programming language – which usually is first translated to some variant of guarded command language before translation to a transition system. In such a scenario, the model checking after receiving the source code (of a high level programming language or a variant of guarded command language) and the property in question, slices the program according to the elements that are of importance in the given property before translating the source code into a transition system. As an example of using program slicing to tackle state space explosion in model checking see [HDZ00, DHH+06].

Example 1.2. As an example of such reduction in practice, consider the model in Figure 1.1. In this system, the resources are simply obtained and then released. Here, this is for the sake of succinctness of the examples. Yet, this system can very well be thought of as the result of slicing a system for checking whether deadlocks exist or not. As what the system does with the resources after it acquires them has no effect on whether the system is in deadlock situation, the unnecessary states between the state where the process owns both resources and the state where it is back in standby state can be simply neglected by a slicing algorithm, resulting in a system as small as depicted in Figure 1.1 for analysis. ■

Partial Order Reduction The nature of state space explosion has its root in parallel running of different processes (or components). Therefore, it often happens that in the transition system of a compound system some different paths are just different interleavings of parallel transitions. In some cases, i.e., where the interleaved transitions are independent, the outcome does not depend on the order of taking these transitions and thus can be abstracted. Partial order reduction approaches analyze the paths of the system and determine which paths of transitions can be considered equivalent; in which case model checking algorithm would need to only examine one, lowering the number of states that should be assessed in practice. These approaches, inspired by [Lip75] and [Maz87], have been widely investigated and adopted in different systems. For further reading on this subject, refer to [Pel93, HP94, Pel96, GP93, GvLH+96, Val91b, Val91a, KLY02, BP02].

Example 1.3. As an example, consider the system in Figure 1.2. In this system, transitions that correspond to requesting and acquiring different resources are independent. In

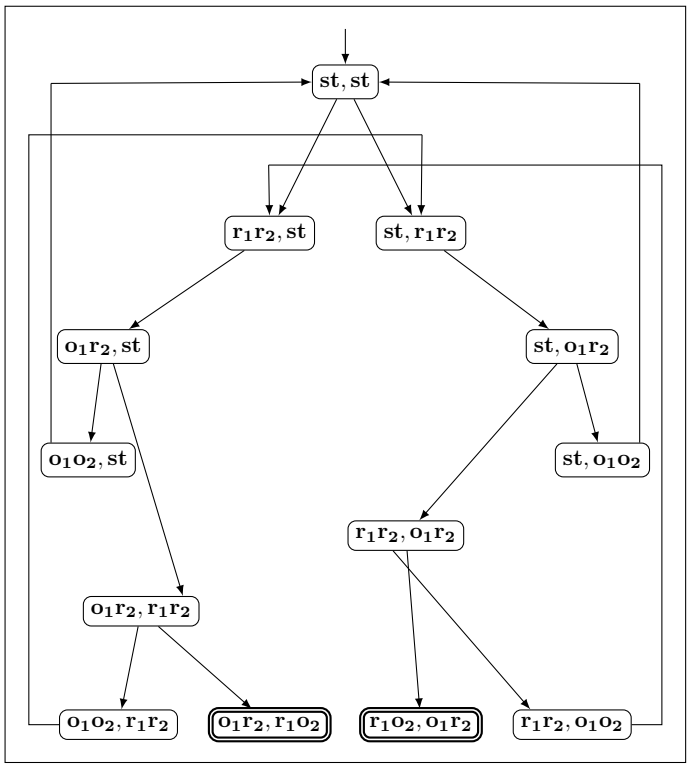


Figure 1.3: The transition system in Figure 1.2 reduced by a path based reduction approach.

other words, it does not matter which one makes the request first or which resource is first allocated and which one afterwards, what matters is that the system is never in a deadlock situation. Thus, in stead of considering the system in Figure 1.2, we can consider a system as simple as the system in Figure 1.3.

Please note that Figure 1.3 is only for the sake of illustration and in practice these techniques usually do not remove states and rather make changes in the system graph (or its representation) to assist the algorithms responsible for checking properties only explore a fragment of the state space. ■

Yet, one should note that such a reduction depends on the property being examined. Here, we have considered that the property being examined is that the system is never in a deadlock situation. Although, if we hypothetically want to check that the process that makes the first request is the one whose request is first acknowledged, the reduction for which the result is depicted in Figure 1.3 would not be useful, as we are interested in the order of the requests being acknowledged. Still, in such a situation, we do not care which resource is first allocated and which afterwards, hence, path based reduction would not be totally useless but perhaps less effective.

1.2.1 Symmetry Reduction

Symmetry is a well known and widely used state space reduction technique. Which utilizes the symmetry of the processes running in parallel to reduce their size. Symmetry occurs when there are identical processes in the system that are run in parallel. As an example, consider a hardware system with n processors.

Although, it should be noted that the sole identity does not by any means guarantee symmetry as interaction with the rest of the system is also of crucial importance in determining existence of symmetry as well. For instance, a system with n processors would only be symmetric if the processors are used symmetrically, e.g., for processing an instance of input, one of them is chosen non-deterministically to carry out the processing operations. On the other hand, if, hypothetically, each processor is designated to interact with different parts of the system – e.g., one only for graphical processing, one solely dedicated to handle network throughput, etc. – symmetry would not exists in spite of the existence of identical components of the system.

Example 1.4. As an example, consider the system in Figure 1.2. In this system, there are two identical processes running in parallel. A quick inspection of the model should unravel the symmetry of this model. In particular, there, when we are interested to know if it is possible for the system to end up in a deadlock, we don't care which process holds which resource or has made the request but we want not to have the situation in which each process owns a resource and is waiting for the other one. In other words, when one process is in state q_1 and the other is in state q_2, we don't care which one is in state q_1 and which one is in q_2. Therefore, we can simply neglect the order of states in each pair, which results in a system depicted in Figure 1.4. ■

Nevertheless, it should be noted that using symmetry – as it dismisses the order of state of individual symmetric processes in the states of the composite system – would not allow for checking of properties that depend on the order of states of individual symmetric states.

In the literature, symmetry is defined with the help of the notion of symmetry groups. In short, symmetry groups can be seen as sets of bijections over a set – here the set of states of the transition system – with some conditions on them. Put simply, a bijection f

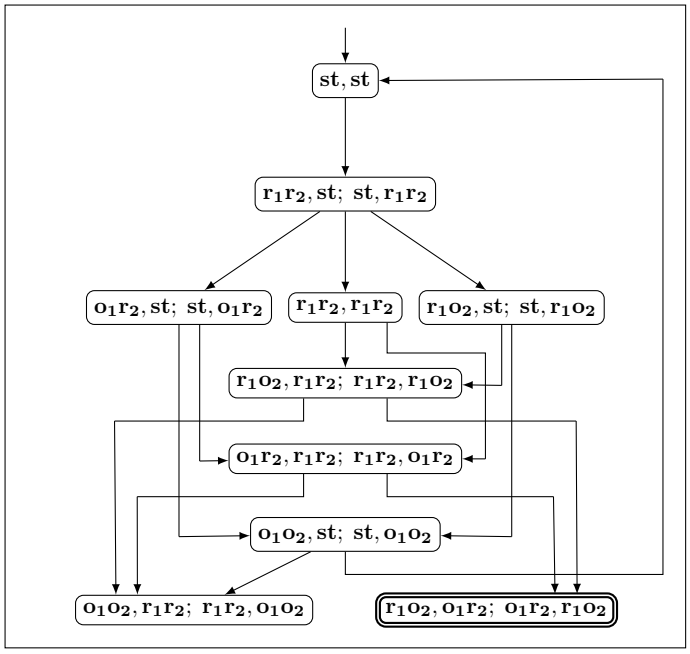


Figure 1.4: The transition system in Figure 1.2 reduced based on symmetry of processes.

is a symmetry permutation if for any pair of states q_1 and q_2 we have, there is a transition from q_1 to q_2 if and only if there is a transition from $f(q_1)$ to $f(q_2)$. We will later elaborate on these notions. For further reading on the subject, the reader is referred to [CEFJ96, CJ95, GS05, ID96, CEJS98, ES97, Ios02, MDC06, SG04, EW03, ET03, DM05, DMP07, DM06, KNP06, BDE+12]. In particular, [MDC06], provides a survey on the subject.

There also has been some efforts to broaden the domain where symmetry can be (more effectively) used. For instance, [SG04], discusses how by adding annotations to symmetry reduced model, one can check also properties that distinguish symmetric processes by unfolding the model on-demand. Emerson. et.al., in [EW03] show how reducing the structure before symbolically representing it helps dramatically reduce the size of symbolic representation. This is achieved by making use of *generic representatives* – counting the number of processes in each state (of the state space of individual symmetric components). In [ET03], on the other hand, the authors discuss how slight changes in the definition of symmetry can help enabling this approach for systems where symmetry is not present by the standard definition while a *near symmetry*, as they call it, is present and can be utilized to reduce the state space of the model.

1.3 Methodology

Introduced in [BSAR06], constraint automata are a type of transition system which are endowed with data-flow locations that mediate data-flow in and out of constraint automata. Moreover, each transition is endowed with a number of constraints over the data-flow throughout constraint automata. We will give a formal definition of the notion of constraint automata in Chapter 3. Yet, for further details on constraint automata, the reader is kindly

referred to [Arb04, BBKK09a, BBKK09b, BSAR06, BKK11, BB08, KB07, Klü12].

For instance, consider the case of a synchronous channel expressed in constraint automata. Such a channel should simply replicate the value of its input to its output at the same time as it receives it. The constraint automata of such a channel would only have assigned input data-flow location, A, and an output data-flow location, B, as well as a single state, q, and a single transition, from q to q, which puts the constraint "the data on A should be the same as the data on B", on the data-flow locations through that transition.

The idea of data-flow locations facilitates combination of constraint automata as components that run in parallel. In other words, an output data-flow location of a constraint automata can be joined to an input data-flow location of another constraint automaton to form a medium for data transfer from one constraint automaton to the other. In turn, any data-flow through a data-flow location that is joined with another data-flow location of another constraint automata should have the agreement of both constraint automata.

In practice, as expected, the combination of constraint automata produces an exponential number of states in the number of components (constraint automata) being joined together.

In this thesis, we are going to define symmetry for constraint automata and show that the result of symmetry reduction is bisimilar – bisimilar transition systems have equivalent behavior – to the original constraint automata; if only those properties are considered that do not depend on the order of symmetric components.

Later, we will up-lift the notion of symmetry to the level of components (individual constraint automata that are symmetric) and discuss how such component symmetry can be confirmed and how reduction can be performed.

In a third step, we will discuss a specific formation of components, named *three tier symmetry*, where symmetry can be, in many cases if not most, efficiently confirmed by only investigating a small fragment of the system. This is done by separating components in three tiers, the symmetry tier, where symmetric components are, glue tier which provides an intermediate medium for data-flow from symmetric components tier to the rest of the system (third tier). We will show that symmetry of such a system can be efficiently (in many cases) confirmed by only examining the glue tier.

In the second part of the thesis, we will discuss how this approach can be employed in the Vereofy model checker. Vereofy is a model checking tool based on constraint automata. It utilizes two languages, CARML (constraint automata reactive module language) and RSL (Reo scripting language), to express models. CARML provides a low level specification of constraint automata which is a form of guarded command language, while, RSL provides a facility for composing constraint automata by specifying which data-flow locations should be joined to form data-flow lines. We will employ the three tier symmetry approach by considering symmetric components specified in CARML while RSL is used to provide the glue tier. There, we will discuss a number of exemplary symmetry scenarios where different glue tiers can be considered to provide an efficient dealing with symmetry. For further details on Vereofy and CARML and RSL specification languages refer to [BKKa, BKKb, Arb04, BSAR06, BBKK09a, KB07].

1.4 Thesis Structure

The rest of this thesis is structured as follows. In Chapter 2 we will introduce the notions of group theory and discuss how they can be utilized to capture and reason about symmetry. In Chapter 3, we will give a formal definition of constraint automata, discuss how

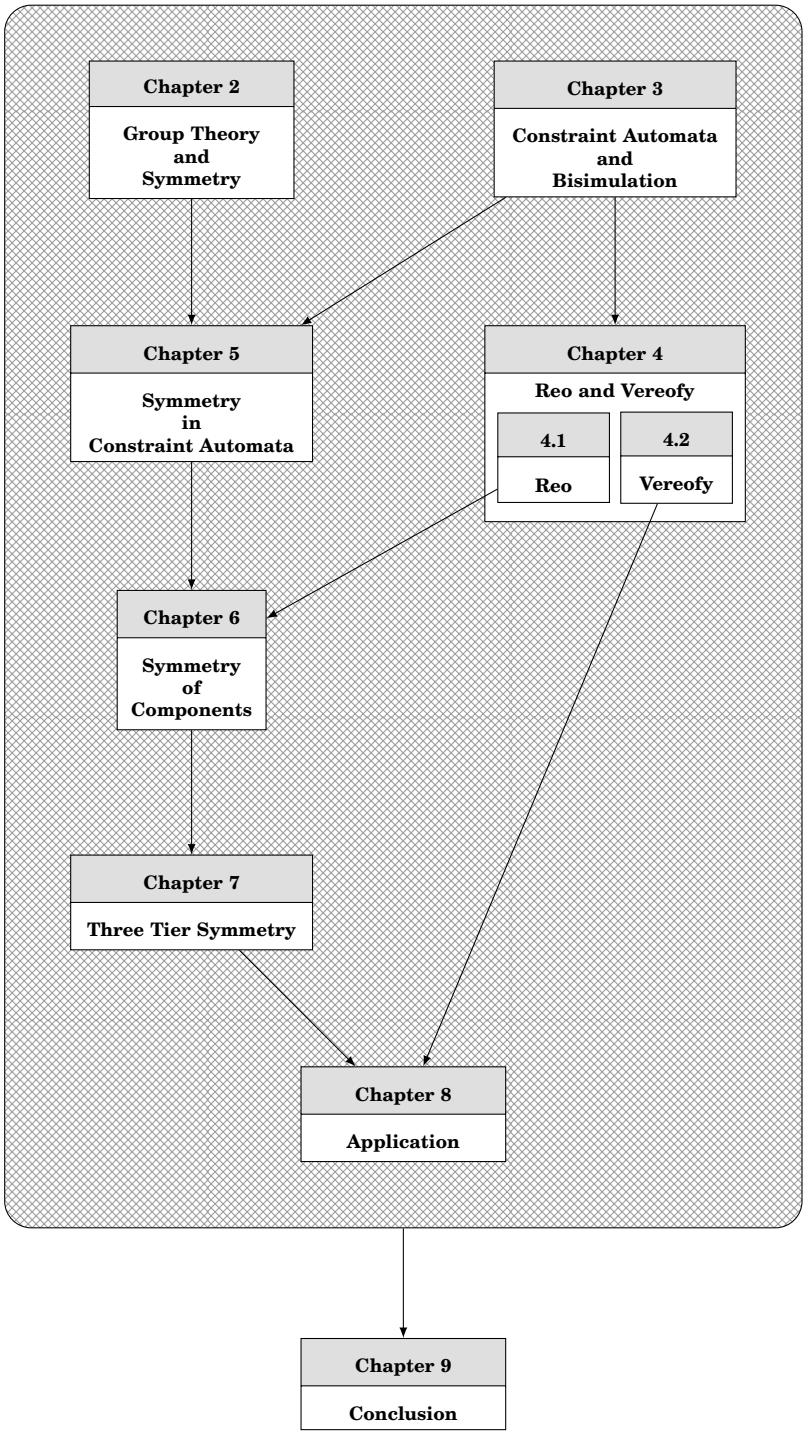


Figure 1.5: The diagram representing the rest of chapters of this thesis and their dependency.

their compositions which allows for building composite systems, and lastly, discuss their bisimilarity.

In Chapter 4, we present the notions of Reo circuits and explain Vereofy and its two languages, CARML and RSL, in more details.

Chapter 5 will introduce the notion of symmetry for constraint automata. Furthermore, in this chapter, we discuss the complexity of checking whether symmetry holds or not, which is **CoNP**-Complete, and show that the result of symmetry reduction is indeed bisimilar to the original constraint automata – if only those properties are considered that are not dependent on the order of symmetric components.

In Chapter 6, we lift the notion of symmetry to the level of components – which are themselves constraint automata. Later, we give a discussion of how without requiring to build the whole composite system and by considering different parts of it at each time symmetry can be confirmed for components; followed by a discussion on how we can compute the result of symmetry reduction without building the whole system.

On the other hand, in Chapter 7, we describe the three tiered symmetry as a type of symmetry over components where the layout of components follows a specific pattern. We will show that considering such restrictions allows for more efficient handling of confirmation and reduction of symmetry of components – by requiring only to reduce only parts of the system as well as needing just to investigate an even smaller portion of the system.

In Chapter 8, we use the results from Chapter 7 to specify a number of scenarios where symmetry can be efficiently handled for models expressed in CARML and RSL that follow those scenarios.

Finally, in Chapter 9, we will give an overview of the contributions of this thesis, discuss possible further research directions and discuss how Vereofy and the implementation of symmetry reduction in it can be improved to cope better with symmetry reduction.

2 Group Theory and Symmetry

In this chapter, we will formally present the well known notion of groups and discuss how they are used to represent and reason about symmetry. This is a well known and practiced method to use groups to reason about symmetry in various contexts including symmetry analysis for model checking. The formalizations in this chapter should be considered as fixing a notation rather than introduction to group theory and the reader should refer to other sources, e.g. [Mil12, MVO04], for a careful consideration of the subject.

2.1 Group Theory

Group theory is an important branch of abstract algebra. Abstract algebra is the generalization of algebra to abstract mathematical structures. In this sense, a group can be seen as the abstraction of very important algebras such as the set of integers together with addition or the set of positive reals together with multiplication. Group theory and in general abstract algebra try to generalize these notions in order to pave the grounds for usage of similar structures in different contexts. An example of such usage is groups of symmetry which are widely used to express and reason about symmetries in various contexts. For further information regarding group theory and groups of symmetry refer to [Mil12, MVO04].

Before presenting the notion of a group, we define two key notions of algebra, i.e., the notion of a set being closed under some operations and the notion of closing a set under some operations. In plain words, a set is closed under some operations if and only if applying any of those operations to any members of that set results in a member of that set.

Definition 2.1 (Closed)**.** Let S be a set and $o : S^k \to S$ a k-ary operation over it. We say S is closed under f if and only if the following holds:

$$\forall s_1, ..., s_k \in S.(o(s_1, ..., s_k) \in S)$$

■

On the other hand, given a set, we can close that set under some operations. In other words, the result of closing a set under some operations is a set that is the smallest – with respect to subset relation – superset of the given set that is closed under those operations.

Definition 2.2 (Closing)**.** Let S be a set and $o_1, ..., o_k$ a number of operations over it. Then, the result of closing S under $o_1, ..., o_k$ denoted by $\langle S \rangle_{o_1,...,o_k}$ is the smallest (with respect to

subset relation) non-empty set T such that:

$$T \neq \emptyset, S \subseteq T \text{ and } T \text{ is closed under } o_1, ..., o_k$$

■

Put simply, a group is a set (usually referred to as underlying set or carrier set) together with two operations called the group operation (binary) and inverse operation (unary). Moreover, there is a specific member of the group called identity element, usually denoted by e. The underlying set of the group is closed under both operations and the group operation is associative. In addition, the result of applying group operation to any member of the group to its inverse (the result of applying inverse operation to it) is the identity element and the result of applying group operation to an arbitrary element of the group and identity element is the that element.

Definition 2.3. A group $\mathcal{G}$ is a mathematical structure $\mathcal{G} = \langle G, *, .^{-1}, e \rangle$, where G is a set, usually called carrier set or underlying set, $*$ is a binary operation over G called group operation, $.^{-1}$ is a unary operation over G called inverse operation, G is closed under $*$ and $.^{-1}$, $e \in G$ is the neutral element of the group and the following group properties hold:

$$\forall_{a,b,c \in G} : (a * b) * c = a * (b * c) \qquad \text{(Associativity)}$$

$$\forall a \in G : a * 0 = 0 * a = e \qquad \text{(Neutral Element)}$$

$$\forall a \in G : a * a^{-1} = a^{-1} * a = e \qquad \text{(Inverse Element)}$$

■

Definition 2.4 (Subgroups)**.** A subgroup of a group $\mathcal{G} = \langle G, *, .^{-1}, e \rangle$ is a group $\mathcal{H} = \langle H, *, .^{-1}, e \rangle$ such that it has the same operations and neutral element and we have $H \subseteq G$. ■

A group is said to be generated by a set if the carrier set of that group is the result of closing that set with group operations. Groups are oftentimes represented by their generating sets.

Definition 2.5 (Generating Set)**.** A group $\mathcal{G} = \langle G, *, .^{-1}, e \rangle$ is generated by a set S if we have $\langle S \rangle_{*,.^{-1}} = G$. ■

Since, we are taking closure of the set in question and by definition the closure of a set can't be empty, we can easily see that the closure of a set under group operations always contains the neutral element of that group.

As we explained earlier, groups are generalization of important algebraic structures such as the set of integers together with addition operation or positive reals (or positive rationals) together with multiplication. Therefore, they can be considered as examples of groups.

Although, other than these simple cases, we have more complicated groups which have their respective and usually quiet important applications. As a more complicated example of groups, we have groups of symmetry whose members are bijections from a set to itself. The fact that function compositions of bijections as well as their inverses (as functions) are also bijections, together with the fact that the identity function of any set is a bijection and is indeed the neutral element of function composition allude to usage of bijections from a set to itself as a group. Which are very useful in definition and reasoning of symmetries in various contexts.

2.2 Symmetry

Given a set S, we can construct a group whose underlying set is a set of bijective functions from S to itself. The group operation and inverse operation of such a group would be function composition and function inversion operations and its neutral element, the identity function over the set S. Such a group is said to be acting on S or a symmetry group of S. The name symmetry group comes from the fact that we can assume bijections to be assigning to each member of S a symmetric (in some sense) member. Symmetry groups are very widely used in different scientific areas, especially in computer science, to express and reason about symmetries in a diversity of contexts.

Definition 2.6 (Groups of Symmetry)**.** Let S be a set. Then, a group acting on S, sometimes called a group of symmetries of S, is any group $\mathcal{G} = \langle G, \circ, .^{-1}, id \rangle$ such that $G \subseteq \{f \mid f : S \to S, f \text{ is a bijection}\}$ is a set of bijective functions, called permutations, from S to itself, $\circ$ is the function composition operation, $.^{-1}$ is the function inverse operation and the permutation id which is the neutral element of group of symmetry is the identity function of set S. ■

For a group of symmetries of a set, orbits of members of that set are the members of the set that are symmetric to that member.

Definition 2.7 (Orbits)**.** Let S be a set, $\mathcal{G} = \langle G, \circ, .^{-1}, id \rangle$ a symmetry group acting on it and $s \in S$ a member of that set. Then, the orbit of s with respect to $\mathcal{G}$, denoted by $\Theta_{\mathcal{G}}(s)$, is the set of all elements that $\mathcal{G}$ indicates as symmetric to s. More formally,

$$\Theta_{\mathcal{G}}(s) = \{g(s) \mid g \in G\}$$

■

Using properties of groups, we can show that orbits partition the underlying sets of the structure. In other words, we can show that given a group acting on a set, we can obtain an equivalence relation whose quotients are the orbits of the given symmetry group.

Definition 2.8. Given a group $\mathcal{G}$ acting on a set S, we define the relation induced by $\mathcal{G}$, denoted by $\equiv_{\mathcal{G}}$. For all $s, t \in S$:

$$s \equiv_{\mathcal{G}} t \text{ if and only if } s \in \Theta_{\mathcal{G}}(t)$$

■

Lemma 2.9. Let $\mathcal{G} = \langle G, \circ, .^{-1}, id \rangle$ be a group acting on a set S and $\equiv_{\mathcal{G}}$ be the relation induced by it. Then, $\equiv_{\mathcal{G}}$ is an equivalence relation on S and its quotient is the set of orbits of $\mathcal{G}$. ■

Proof. We first prove that $\equiv_{\mathcal{G}}$ is an equivalence relation by showing properties of equivalence relations as follows:

Reflexivity:

$$\forall s \in S.(id(s) = s) \wedge id \in G$$

thus,

$$\forall s \in S.\ s \equiv_{\mathcal{G}} s$$

Symmetry:

$$s \equiv_{\mathcal{G}} t \implies s \in \Theta_{\mathcal{G}}(t) \implies \exists g \in G.\, g(t) = s$$

by group properties, we have,

$$g^{-1} \in G$$

and,

$$g^{-1}(s) = t \implies t \in \Theta_{\mathcal{G}}(s)$$

which results in,

$$t \equiv_{\mathcal{G}} s$$

Transitivity:

$$s \equiv_{\mathcal{G}} t \implies s \in \Theta_{\mathcal{G}}(t) \implies \exists g \in G.\, g(t) = s$$
$$t \equiv_{\mathcal{G}} t' \implies t \in \Theta_{\mathcal{G}}(t') \implies \exists g' \in G.\, g'(t') = t$$

by group properties, we have,

$$g' \circ g \in G$$

and,

$$(g' \circ g)(s) = t' \implies t' \in \Theta_{\mathcal{G}}(s)$$

which results in,

$$s \equiv_{\mathcal{G}} t'$$

Since, for any $s \in S$ the equivalence class of s with respect to $\equiv_{\mathcal{G}}$, denoted by $[s]_{\equiv_{\mathcal{G}}}$, and its orbit with respect to $\mathcal{G}$ are uniquely defined, and always exist, we only need to show that these two sets are equal. Which is trivial from Definition 2.8. The following is a restatement of the condition of Definition 2.8 which makes it clearer:

$$\forall s, t \in S.((s \in [t]_{\equiv_{\mathcal{G}}}) \Leftrightarrow (s \in \Theta_{\mathcal{G}}(t)))$$

□

3 Constraint Automata and Bismulation

Constraint automata, introduced in [BSAR06] and inspired by the exogenous coordination language Reo presented in [Arb04], are a type of transition system where transitions are subject to satisfying a number of constraints associated with them, which restrict data-flow throughout the constraint automata by constraining activation and valuation of data-flow locations – entities which represent mediums through which data flows. There are three kinds of data-flow locations, input data-flow locations, output data-flow locations and internal data-flow locations; which are respectively, to mediate data flow, from "outside world" to the constraint automata, from constraint automata to the "outside world" and internally. Constraint automata can be used to describe the behavior of components of the system – the whole system can also be considered a component. In this regard, the term "outside world" refers to the rest of the components of the system or the environment which system works in.

Considering constraint automata as components of the system, there should be a mechanism for composing these components to get bigger components – or the whole system. This mechanism, called *joining*, uses data-flow locations to compose constraint automata. That is, output data-flow locations of one constraint automaton are "connected" to input data-flow locations of the other constraint automaton to form some sort of "data communication lines" through which data can flow from one component to the other. In turn, the data-flow locations that are used to join the two components have been connected to some other data-flow location to form a communication line to mediate data flow which is now considered internal data-flow and hence, such data-flow locations would be considered internal data-flow locations of the result of joining the two constraint automata. On the other hand, the flow of data through these communication lines should be in compliance with both constraint automata of both ends of the communication line. This facilitates synchronization throughout the network of constraint automata that are composed to form the system.

In addition, in a constraint automata, we can *hide* a data-flow location. The operation of hiding a data-flow location, removes a data-flow location from a constraint automata, while, changes the constraints of the transition in such way that existence of some non-deterministically chosen value is assumed to be on that data-flow location. As a result, the activity of, and the data-flow through, the hidden data-flow location is hidden from the outside world. Hence, hiding a data-flow location, in the case of internal data-flow locations, makes it hidden from the outside world, and in the case of input or output data-flow locations makes it unavailable for other components of the system to use it to connect to that component. This operation can be seen as hiding implementation details when

observing a model with a higher level of abstraction.

Additionally, two constraint automata are said to be *bisimilar*, if they can simulate the behavior of one another. In other words, if for any state in a constraint automaton, there is (at least) a state in the other constraint automaton which behaves similarly, i.e., has transitions to and from states which are bisimilar to the states where the other state has transitions to and from them. Among other notions of equivalence for transition systems, bisimulation has particular importance, as two bisimilar transition systems are guaranteed to have the same temporal properties (expressible in LTL, CTL and CTL* temporal logics). Hence, the result of model checking (satisfaction or rejection) of a property expressed in temporal logic for two bisimilar transition systems is guaranteed to be the same. This is of particular interest in this thesis as we will show the result of symmetry reduction is bisimilar to the original constraint automaton – if only those properties are considered that are preserved under symmetry – which means the result of reduction can be safely used for model checking instead of the original constraint automaton.

In this chapter, we will give the formal definition of constraint automata and well as their join and hide operations. Afterwards, we will give the formal definition of their basimilarity.

In the rest of this thesis, we will use notations very similar to that of [BKK11] and [BBKK09a]. In this chapter, we will formally and concisely introduce these notions. For further reading on constraint automata and their bisimulation, an interested reader can always refer to [BKK11, BBKK09a, BBKK09b, BB08]. For further reading on bismulation and other measures of equivalence for transition systems in general, the reader is kindly referred to [Mil80, Par81, vG93, vG90] and [BCG88].

3.1 Constraint Automata

Constraint automata are a type of transition system in which transitions are labeled with a set of active data-flow locations as well as constraints that specify what data values those active data-flow locations can have throughout that transition. Before introducing constraint automata, we will first introduce the form of constraints that are going to be used in the definition of constraint automata.

Definition 3.1 (Data Constraint)**.** Let $\mathcal{N}$ be the set of data-flow locations and $\mathfrak{D}$ be the set of values that can be used in component communications. Then, the following grammar describes the language of data constraints.

$$\begin{array}{rcll} \varphi & ::= & true & \\ & | & d_A = a & \text{where } A \in \mathcal{N}, a \in \mathfrak{D} \\ & | & d_A = d_B & \text{where } A, B \in \mathcal{N} \\ & | & \varphi_1 \vee \varphi_2 & \\ & | & \neg\varphi & \end{array}$$

Here, d_A denotes the value of data-flow location A. Furthermore, the set of all data constraints over a set of data-flow locations $\mathcal{N}$ is denoted by $\mathfrak{Dcon}(\mathcal{N})$ ■

Please note that, $\neg$ and $\vee$ form a complete set of connectives for propositional logic and hence, other connectives, e.g., $\wedge, \rightarrow$, etc., are derived and used as usual.

Definition 3.2 (Constraint automata)**.** A constraint automaton is a tuple

$$\mathcal{A} = \langle Q, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out} \rightarrow, Q_0 \rangle$$

where

- Q is a set of states
- $\mathcal{N}$ is a set of data-flow locations
- $\mathcal{N}_{in}, \mathcal{N}_{out} \subseteq \mathcal{N}$ with $\mathcal{N}_{in} \cap \mathcal{N}_{out} = \emptyset$
- $\rightarrow \subseteq Q \times 2^{\mathcal{N}} \times \mathfrak{Dcon}(\mathcal{N}) \times Q$ is the transition relation
- $Q_0 \subseteq Q$ is the set of initial states

■

In the rest of this thesis, we will usually abbreviate transitions of the form $(q, N, g, q') \in \rightarrow$ as $q \xrightarrow{N,g} q'$.

When the current state of the automaton changes as a result of a transition being performed, the data constraints of the transition define a notion of concurrent data flow to and from the automaton and outside world. As an example, the transition $q \xrightarrow{\{A,B\}, d_A = d_B} q'$, where $A \in \mathcal{N}_{in}$ and $B \in \mathcal{N}_{out}$, the underlying concurrent data flow is: "transition from q to q' while channeling the data value at data-flow location A to data-flow location B".

In order to define the semantics of constraint automata we first formally define the concept of concurrent I/O operations to represent the flow of data throughout data-flow locations during transitions of constraint automata.

Definition 3.3 (Concurrent I/O operation)**.** Let $\mathcal{N}$ be a set of data-flow locations and $\mathfrak{D}$ be the set of values that these data-flow locations can take. Then, a concurrent I/O operation, usually abbreviated as CIO, is a partial function $c : \mathcal{N} \rightarrow \mathfrak{D}$. For a data-flow location A, we use $c(A) = \perp$ to indicate that there is no value for A assigned by concurrent IO operation c.

We denote the set of all concurrent I/O operations defined over the set of data-flow locations $\mathcal{N}$ as $\mathfrak{Cio}(\mathcal{N})$. The set $active(c) = \{A \in \mathcal{N} \mid c(A) \neq \perp\}$ denotes the active data-flow locations in the concurrent I/O operation c. In addition, we use ε to denote the unique concurrent I/O operation that does not assign any value to any data-flow location, i.e., $active(c) = \emptyset$. ■

In order to define the semantics underlying constraint automata, we need to define the semantics of data constraints. We say a CIO models a data constraint, if it is consistent with that constraint.

Definition 3.4 (Constraint Modeling and Equivalence Relation)**.** Let $\mathcal{N}$ be the set of data-flow locations, $c \in \mathfrak{Cio}(\mathcal{N})$ and $g \in \mathfrak{Dcon}(\mathcal{N})$ be a data constraint over $\mathcal{N}$. Then, c is a model of g, denoted by $c \models g$, if and only if:

$$
\begin{array}{lcll}
c & \models & true & \\
c & \models & d_A = d & \text{if} \quad c(A) = d \\
c & \models & d_A = d_B & \text{if} \quad c(A) = c(B) \\
c & \models & g_1 \vee g_2 & \text{if} \quad c \models g_1 \text{ or } c \models g_2 \\
c & \models & \neg g' & \text{if} \quad c \not\models g'
\end{array}
$$

And for $g' \in \mathfrak{Dcon}(\mathcal{N})$, g is equivalent with g', denoted by $g \equiv g'$, if and only if:

$$\forall c \in \mathfrak{Cio}(\mathcal{N}).\ c \models g \Leftrightarrow c \models g'$$

■

Definition 3.5 (CIO Consistency)**.** Let $\mathcal{N}$ be the set of data-flow locations, $N \subseteq \mathcal{N}$ be a set of data-flow locations and $g \in \mathfrak{Cio}(\mathcal{N})$ be a data constraint over $\mathcal{N}$. Then, $Cio(N, g)$ denotes the set of all CIOs that are consistent with N (as active data-flow locations) and g (as data constraint):

$$Cio(N, g) = \{c \in \mathfrak{Cio}(\mathcal{N}) \mid active(c) = N \wedge c \models g\}$$

■

As discussed earlier, constraint automata have been introduced with the intuition of their application in modeling and verification. Therefore, we define paths in the constraint automata which would allow for temporal logic model checking, as they provide the means to analyze the chronology of state transitions within a constraint automata.

Definition 3.6 (Paths)**.** In a constraint automaton $\mathcal{A} = \langle Q, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out}, \rightarrow, Q_0 \rangle$, an execution path is an infinite sequence:

$$\pi = q_0 \xrightarrow{c_0} q_1 \xrightarrow{c_1} q_2 \xrightarrow{c_2} \dots$$

where, $\forall i \geq 0 : q_i \in Q, c_i \in \mathfrak{Cio}(\mathcal{N})$ and there exists a transition $q_i \xrightarrow{N,g} q_{i+1}$ such that $c_i \in Cio(N, g)$. ■

Please note that in model checking contexts, it is assumed that paths are infinite and in case of blocking states, transitions are added to the system from those states to some trap states to make sure all paths in the transition system can be extended to infinite ones. As it is not directly related to the topic of this thesis, we will not formally introduce model checking for constraint automata and stop at defining paths to give an intuition. The interested reader is kindly referred to [BKK11, BBKK09a, Klü12].

3.1.1 Composition of Constraint Automata

As we said earlier, constraint automata can represent the components of the system and therefore we require a mechanism to compose the automata of individual components to get bigger components or the whole system. In this process, called joining, data-flow locations play an important role, as they provide lines of communication for components.

In the process of joining two constraint automata, we assume that, there are no data flow locations shared by the two automata other than those that are an input data-flow location for one and output data-flow location for another. As a result, the process of composing two constraint automata is unifying those shared data-flow locations. In doing so, we consider three different cases. Cases where only one the constraint automata is undergoing a transition and the other is not – which is only possible if the transition does not have any of the shared data-flow locations active, as though they should both agree on its value. Case where they are each are undergoing a transition (at the same time) – in which case both transitions should agree on the activation and values of shared data-flow locations. The result of joining two constraint automaton is referred to as product automaton and is defined in the followings. In Definition 3.7, these cases can be seen as conditions (3.1), (3.2) and (3.3). Condition (3.2) and (3.3) represent cases where $\mathcal{A}_1$ is making a transition while $\mathcal{A}_2$ is not and cases where $\mathcal{A}_2$ is making a transition while $\mathcal{A}_1$ is not, respectively. While, condition (3.1) corresponds to cases where both $\mathcal{A}_1$ and $\mathcal{A}_2$ are making a transition. It is worth noting that this latter condition (Condition (3.1)) can be satisfied in two ways. Either, if they share some active shared data-flow locations – in which case they should agree upon their values –, or, if they don't have any active shared data-flow locations in common.

Definition 3.7 (Product Automaton)**.** Let $\mathcal{A}_i = \langle Q_i, \mathcal{N}_i, \mathcal{N}_{in}^i, \mathcal{N}_{out}^i \rightarrow_i, Q_0^i \rangle$ for $i \in \{1, 2\}$ be two constraint automata, where $\mathcal{N}_1 \cap \mathcal{N}_2 = (\mathcal{N}_{in}^1 \cap \mathcal{N}_{out}^2) \cup (\mathcal{N}_{out}^1 \cap \mathcal{N}_{in}^2)$. Then, the product automaton, $\mathcal{A}_1 \bowtie \mathcal{A}_2$, is defined as follows:

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = \langle Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, \mathcal{N}_{in}, \mathcal{N}_{out} \rightarrow, Q_0^1 \times Q_0^2 \rangle$$

where,

$$\begin{aligned} \mathcal{N}_{in} &= (\mathcal{N}_{in}^1 \cup \mathcal{N}_{in}^2) \setminus (\mathcal{N}_1 \cap \mathcal{N}_2) \\ \mathcal{N}_{out} &= (\mathcal{N}_{out}^1 \cup \mathcal{N}_{out}^2) \setminus (\mathcal{N}_1 \cap \mathcal{N}_2) \end{aligned}$$

and

$$\frac{q_1 \xrightarrow{N_1, g_1} q_1',\ q_2 \xrightarrow{N_2, g_2} q_2',\ N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1}{\langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle q_1', q_2' \rangle} \tag{3.1}$$

$$\frac{q_1 \xrightarrow{N_1, g_1} q_1',\ N_1 \cap \mathcal{N}_2 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{N_1, g_1} \langle q_1', q_2 \rangle} \tag{3.2}$$

$$\frac{q_2 \xrightarrow{N_2, g_2} q_2',\ N_2 \cap \mathcal{N}_1 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{N_2, g_2} \langle q_1, q_2' \rangle} \tag{3.3}$$

■

3.1.2 Hiding Data-Flow Locations

There are several different reasons why a data-flow location might be required hidden. For instance, if its activity is of no particular interest seen from outside world, for increasing the level of abstraction of the model, etc. During the operation, the data-flow location being hidden is removed from the constraint automaton and the set of active data-flow locations of every transition while transition constraints are altered to cope with this removal. More specifically, in the constraints of each transition it is assumed that there is a value of data-domain, $\mathfrak{D}$, is present on that data-flow location. Hence, the hidden data-flow location is not observable anymore while its effect is still present in the resulting constraint automaton –as a result of assuming existence of some value.

Definition 3.8 (Hiding)**.** Let $\mathcal{A} = \langle Q, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out}, \rightarrow, Q_0 \rangle$ be a constraint automaton and $A \in \mathcal{N}$ be a data-flow location. Then, the result of hiding data-flow location A from $\mathcal{A}$ is a constraint automaton

$$\exists [A]\mathcal{A} = \langle Q, \mathcal{N} \setminus \{A\}, \mathcal{N}_{in} \setminus \{A\}, \mathcal{N}_{out} \setminus \{A\}, \rightarrow_{\setminus \{A\}}, Q_0 \rangle$$

where:

$$\frac{q \xrightarrow{N, g} q'}{q \xrightarrow{N \setminus \{A\}, \exists [A] g}_{\setminus \{A\}} q'}$$

and

$$\exists [A] g = \bigvee_{d \in \mathfrak{D}} g[d_A / d]$$

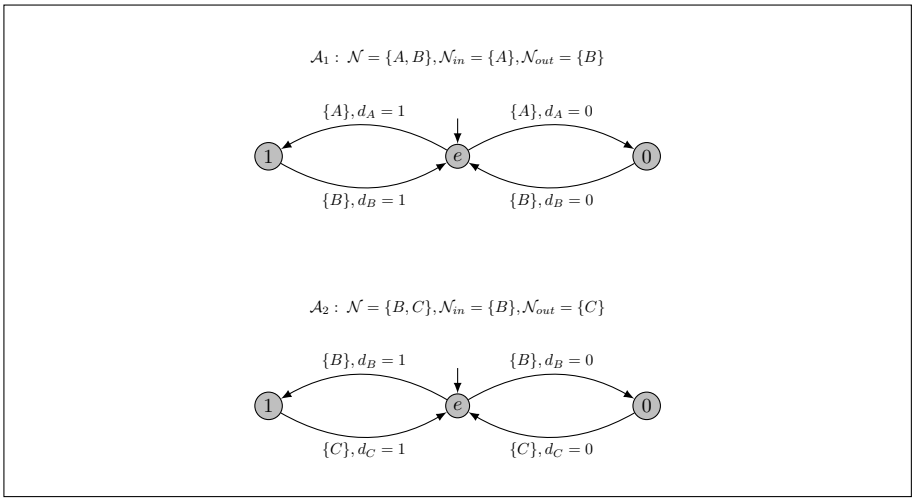


Figure 3.1: Example of two constraint automata (components). See Example 3.9.

where $g[d_A/d]$ is defined as follows:

$$\begin{array}{lcll}
true[d_A/d] & = & true & \\
(g_1 \vee g_2)[d_A/d] & = & g_1[d_A/d] \vee g_2[d_A/d] & \\
(\neg g)[d_A/d] & = & \neg(g[d_A/d]) & \\
(d_A = d_B)[d_A/d] & = & d_B = d & \\
(d_B = d_C)[d_A/d] & = & d_B = d_C & \\
(d_A = d')[d_A/d] & = & true & d = d' \\
(d_A = d')[d_A/d] & = & false & d \neq d' \\
(d_B = d')[d_A/d] & = & d_B = d' &
\end{array}$$

In the rest of this thesis, we will use the short hand notation

$$\exists[A_1, \ldots, A_k]\mathcal{A} = \exists[A_1](\ldots \exists[A_{k-1}](\exists[A_k]\mathcal{A}) \ldots)$$

to denote simultaneous hiding of data-flow locations $A_1, \ldots, A_k$ from $\mathcal{A}$. ■

As an example of constraint automata, join and hide operations see Example 3.9 below.

Example 3.9. Figure 3.1 depicts two constraint automata each of which representing a FIFO queue with one cell – state e representing empty fifo, state 0 and 1 representing fifo containing value 0 and 1 respectively. For $\mathcal{A}_1$ the input data-flow location is A and the output data-flow location is B while for the second, the input data-flow location is B and output data-flow location is C. Hence, the result of joining these two constraint automata – as data-flow location B is shared – would result in a fifo with two cells, depicted in Figure 3.2.

The composite component depicted in Figure 3.2, has a data-flow location B, though, which is neither an input nor an output data-flow location. Data flow through this data-flow location is in fact an internal data-flow – corresponding to internal operation of sending data from first cell to the next. Figure 3.3, depicts the result of hiding this data-flow location – making internal data flow invisible from outside world. ■

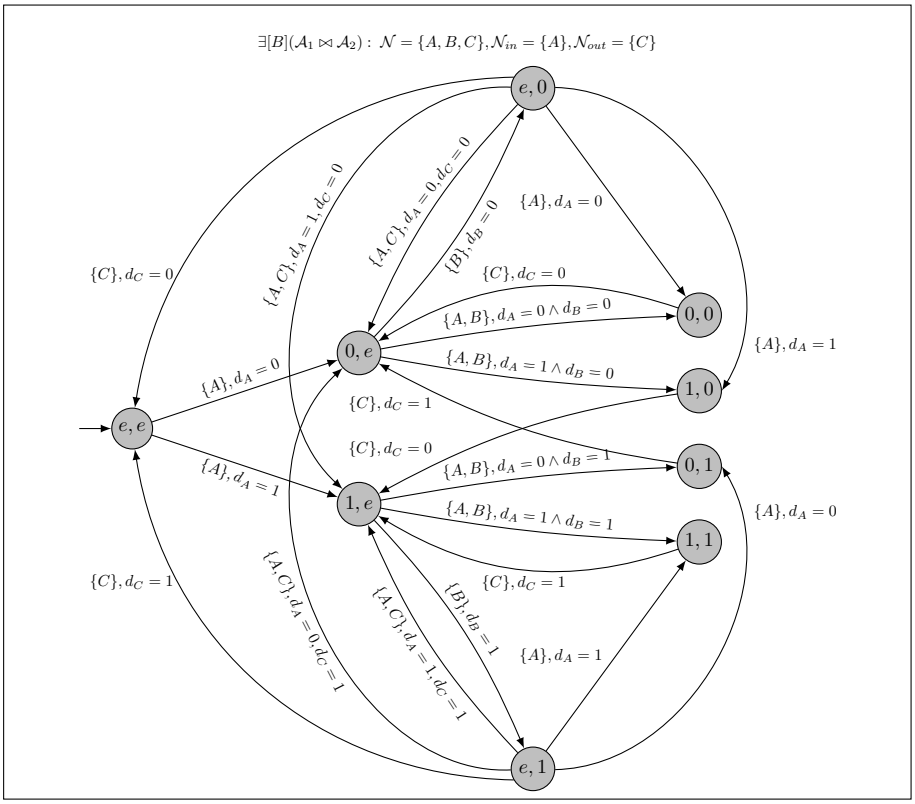


Figure 3.2: The result of joining components in Figure 3.1 – Where, $\mathfrak{D} = \{0, 1\}$. See Example 3.9.

3.2 Bisimilarity of Constraint Automata

Other than checking whether a transition system possesses the properties of the design goal, it can in some cases be necessary to check if two transition systems have similar behavior, e.g., when different abstractions of the same model must be confirmed having similar behavior (approximations of the behavior of the implementation), or when we want to show that the result of a process (like symmetry reduction) preserves the behavior (or certain parts of it) of the original model. There are several different methods of comparison of the behavior of transition systems, bisimulation being one of them. For more details on various methods to compare behavior of transition systems, please refer to [vG90, vG93].

Action-based bisimulariy was first brought into attention independently in [Mil80] and [Par81] which was the inspiration of state based bisimulation (the notion that we present here as bisimulation). In general, two transition systems are said to be bisimilar if they can't be distinguished when only considering their temporal behavior, i.e., the chronology of their state changes. More precisely, for each state in one system there should exist at least one state in the other system such that they have equivalent behavior, i.e., all their transitions are to and from states which in turn have similar temporal behavior.

In particular, bisimulation is of importance because of the fact that two bisimilar tran-

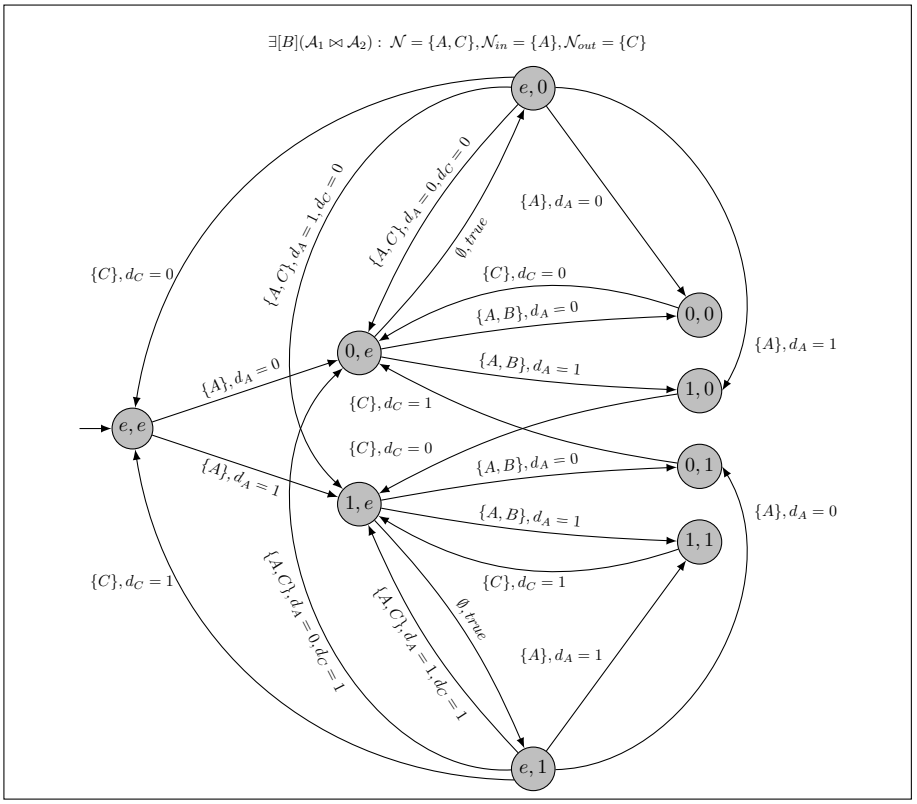


Figure 3.3: The result of hiding data-flow location B from the constraint automata in Figure 3.2. See Example 3.9.

sition systems are guaranteed to satisfy the same properties expressed in LTL, CTL (which are included in CTL*) and CTL* – temporal logics which are used to express temporal properties for model checking purposes – which was shown in [BCG88].

In model checking, instead of inspecting temporal flow of states, temporal flow of *atomic propositions* are considered. *Atomic propositions* are basic characteristics that a state can have. Each state is usually labeled with atomic propositions. This would allow for more concise temporal formula. As an example, consider the constraint automaton of Figure 3.3. In this figure, we can simply label the state (e, e) with *empty*, label all those states of the form (e, i) and (i, e) for $i \in \{0, 1\}$ with *half-full* and label all states of the form (i, j) for $i, j \in \{0, 1\}$ with *full*. Then, we can simply specify the temporal property "the FIFO never goes from *empty* to *full* without first being *half-full*", while without atomic propositions such property would have been longer and a lot more complicated.

On the other hand, since the purpose of defining bisimilarity is to capture constraint automata that simulate the behavior of one another, we will first formally define the concept of *labeled constraint automata* which is the constraint automata endowed with atomic propositions and then use it to define the notion of bisimilarity.

Definition 3.10 (Labeled Constraint Automata)**.** A labeled constraint automaton is a tuple $\mathcal{A} = \langle Q, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out} \rightarrow, Q_0, AP, L \rangle$ where

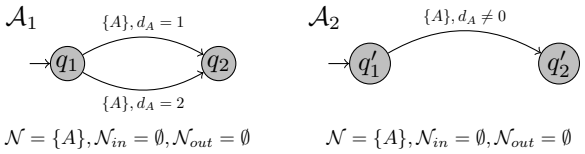
- $\langle Q, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out} \rightarrow, Q_0 \rangle$ is a constraint automaton
- AP is a finite set of atomic propositions (labels) and
- $L : Q \rightarrow 2^{AP}$ is the labelling function

■

In the sequel, we will use names 'constraint automata' and 'labeled constraint automata' interchangeably and only take atomic propositions in account if they are needed.

When defining bisimilarity for constraint automata, one has to be taking constraints into account as well; as for equivalence of behavior, one should consider equivalence of underlying data constraints as well. Basically, not only do we want similar states to be transitioning to similar states but also these transitions should be through possible with the same data-flow. For further clarity, consider Example 3.11. In this example, the pair of states q_1 and q'_1 are bisimilar, yet considering only the syntax of data constraints and transition relation of the constraint automata it can not be observed.

Example 3.11. Consider two constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$:



Obviously, assuming data-domain is $\mathfrak{D} = \{0, 1, 2\}$, the two constraint automata are bisimilar, yet this can not be concluded if only the transition relation and the syntax of data constraints are considered.

■

Therefore, we will introduce the notion of constraints between two states as the general constraint that make state transitions viable. In Example 3.11, for both state pairs (q_1, q_2) and (q'_1, q'_2) the constraints between states of each pair, "$d_A = 1 \vee d_A = 2$" and "$d_A \neq 0$", respectively, are equivalent. In other words, they are both only satisfied by CIOs that assign a value of 1 or 2 to data-flow location A. The notion of constraints between is formally defined as follows.

Definition 3.12 (Constraints Between Two States)**.** Let $\mathcal{A}$ be a constraint automaton with the set of states $Q_{\mathcal{A}}$, the set of data-flow locations $\mathcal{N}_{\mathcal{A}}$ and the transition relation $\rightarrow_{\mathcal{A}}$. Then, the set of Constraints between two states $q, q' \in Q_{\mathcal{A}}$ for a set of data-flow locations $N \subseteq \mathcal{N}_{\mathcal{A}}$ denoted by $\mathfrak{C}^{\mathcal{A}}_{q,q'} : \mathcal{N}_{\mathcal{A}} \rightarrow \mathfrak{D}\mathtt{con}(\mathcal{N}_{\mathcal{A}})$ is defined as follows:

$$\mathfrak{C}^{\mathcal{A}}_{q,q'}(N) = \bigvee_{q \xrightarrow{N,g}_{\mathcal{A}} q'} g$$

Please note that, in case there is no transition such transition as $q \xrightarrow{N,g}_{\mathcal{A}} q'$, $\mathfrak{C}^{\mathcal{A}}_{q,q'}(N)$ would be equal to the empty generalized disjunction which is $false$. ■

The notion of bisimilarity for constraint automata was firstly defined in the introductory paper of the constraint automata, [BSAR06]. The concept of bisimilarity presented here is very similar to that of [BSAR06]. It is only that here, we have incorporated atomic

propositions in the definition of bisimilarity, as it is oftentimes done (e.g., Chapter 7 of [BK08]), so that we can discuss properties that are preserved under symmetry reduction. This is due to the fact that, as we will see, there is some sort of bisimulation between a constraint automatan and the result of its reduction with respect to symmetry.

Definition 3.13 (Bisimilarity)**.** Let $\mathcal{A}_i = \langle Q_i, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out} \rightarrow_i, Q_0^i, AP, L_i \rangle$ for $i \in \{1, 2\}$ be two labeled constraint automata with the same data-flow locations and atomic propositions. Then, $\mathcal{A}_1$ and $\mathcal{A}_2$ are bisimilar if and only if there exists a relation $\sim \subseteq Q_1 \times Q_2$, usually referred to as bisimulation relation, such that:

(A) $\forall q_1 \in Q_0^1 \exists q_2 \in Q_0^2.\ q_1 \sim q_2$ and $\forall q_2 \in Q_0^2 \exists q_1 \in Q_0^1.\ q_1 \sim q_2$

(B) $q_1 \sim q_2$ then, the followings hold:

(1) $L_1(q_1) = L_2(q_2)$

(2) $\forall q_1' \in Q_1 \exists q_2' \in Q_2. \left(q_1' \sim q_2' \wedge \left(\forall N \subseteq \mathcal{N}.\ \mathfrak{C}^{\mathcal{A}_1}_{q_1, q_1'}(N) \equiv \mathfrak{C}^{\mathcal{A}_2}_{q_2, q_2'}(N)\right)\right)$

(3) $\forall q_2' \in Q_2 \exists q_1' \in Q_1. \left(q_1' \sim q_2' \wedge \left(\forall N \subseteq \mathcal{N}.\ \mathfrak{C}^{\mathcal{A}_1}_{q_1, q_1'}(N) \equiv \mathfrak{C}^{\mathcal{A}_2}_{q_2, q_2'}(N)\right)\right)$

■

In this thesis, we have not given any details on model checking problems; and only referred the reader to relevant material. Yet, it should be noted that in model checking scenarios, only states are considered that are reachable from some initial state. As a result, those states that are not reachable from some initial state do not impose any effect on the outcome of model checking tasks. Hence, the definition of bisimilarity is defined to capture only states that are reachable from some initial state – in Definition 3.13, the bisimulation relation is only required to have bisimilar initial states while the second and third conditions of that definition guarantees that for any reachable state in one constraint automaton there is a bisimilar one in the other constraint automaton. Furthermore, here, for the sake of simplicity, we have assumed any transition system in general – and hence constraint automata – have only those states that are reachable from some initial state.

Baier et.al, in [BSAR06], show that the bisimulation problem for constraint automata is **CoNP**-hard. This is due to the fact that for confirming bisimilarity, we should show the underlying constraints for reachability of corresponding pairs of states are equivalent. This requirement, is a very powerful one and would easily allow for confirmation of equivalence of propositional formula, which is well known to be **CoNP**-Complete. For further details on bisimilarity of constraint automata and its complexity, please refer to [BSAR06]. For further reading on the subject of computational complexity, refer to [AB09].

For further information on bisimulation, in general, see [BK08, BBKK09a, BSAR06, Mil80, Par81, vG90, vG93, BCG88].

4 Reo and Verofy

Reo is an exogenous channel based coordination language introduced in [Arb04]. In Reo, a model (Reo circuit) consists of a number of components for which Reo nodes (points of distribution of data) and channels provide communication. Baier et. al., in [BSAR06], introduced constraint automata to form semantics for Reo circuits. In such a framework, constraint automata are used to describe components as well as Reo nodes and channels – see [BKK11]. Vereofy (see [BKK11, BBKK09b, BKKa, BKKb, Klü12]) is a model checking tool that is based on this idea. Here, we briefly describe the the parts of Reo circuits and Vereofy that we are going to need in this thesis. We will consicely describe the notions of Reo circuits and Vereofy model checking tool together with its model specification languages CARML (Constraint Automata Reactive Module Language) which is used to describe components and RSL (Reo Scripting Language).

4.1 Reo

Reo, introduced in [Arb04], is a channel based exogenous coordination language devised for coordination of components of a system. In this framework, *Channels* and *nodes* are the basic concepts forming the core of coordination language. Put simply, a channel is a medium through which data is transmuted in Reo circuits. While nodes are data concentration points where it is decided how data are distributed over different components and channels.

Constraint automata, as they were initially introduce to this end (see [BSAR06]), are a well-suited candidate for underlying semantics of Reo coordination language. In such a modeling framework, constraint automata are utilized to express the semantics of components as well as channels and nodes. Indeed, Model checking tool Vereofy, which we will later discuss in more details, follows this idea.

4.1.1 Channels

In Reo, each component is equipped with a number of channel ends which are points of their communication to and from the outside world (the rest of the circuit). Each channel end can either be a *source* channel end or a *sink* channel end, which are points through which data flows into and out of a channel respectively. From this perspective, Reo channels are components that have exactly two channel ends. Reo has a library of predefined channels with different underlying semantics, this collection can be complemented with user defined channels to provide a suitable coordination for the system at hand.

Figure 4.1 shows some exemplary channels of Reo library. The first channel in this figure is a *synchronous channel*, which has two channel ends represented by small hollow circles

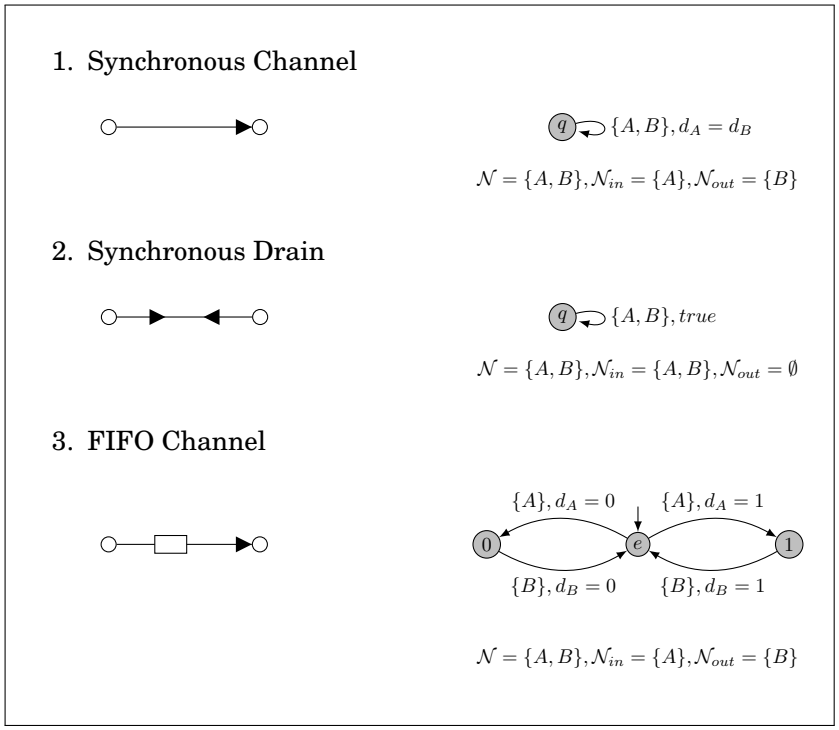


Figure 4.1: A few exemplary Reo channels from Reo's builtin library of channels.

at either side in its graphical representation. This channel simply replicates the data on its source channel end (represented by data-flow location A in the corresponding automaton) to its sink channel end (represented by data-flow location B in the corresponding automaton) at the same time as it receives it. The second channel is a *synchronous drain*, which has two source channel ends. This channel would allow data-flow only if both its source channel ends (represented by data-flow locations A and B in its corresponding automaton) are active at the same time.

The third channel in Figure 4.1, i.e., *FIFO channel*, on the other hand, is slightly more complex. This channel has a source and a sink channel end, represented by data-flow locations A and B, respectively. When ever it receives a data value through its source channel end, it stores that value, and does not accept any activity on its source channel end until the stored data value is sent out through its sink channel end. The constraint automata depicted in the Figure 4.1 for this channel is depicted under the assumption that data domain $\mathfrak{D} = \{0, 1\}$.

4.1.2 Reo Nodes

Reo nodes, or simply nodes, are points of concentration and distribution of data in a Reo circuit. Nodes can be thought of as a collection of channel ends, each of which with a specific distribution semantics. In this thesis, we are going to assume two types of nodes, *replicate nodes* (depicted by ●) and *route nodes* (depicted by ⊗). On the other hand, a node can be a *source node*, if it only contains source channel ends, a *sink node*, if it only contains sink

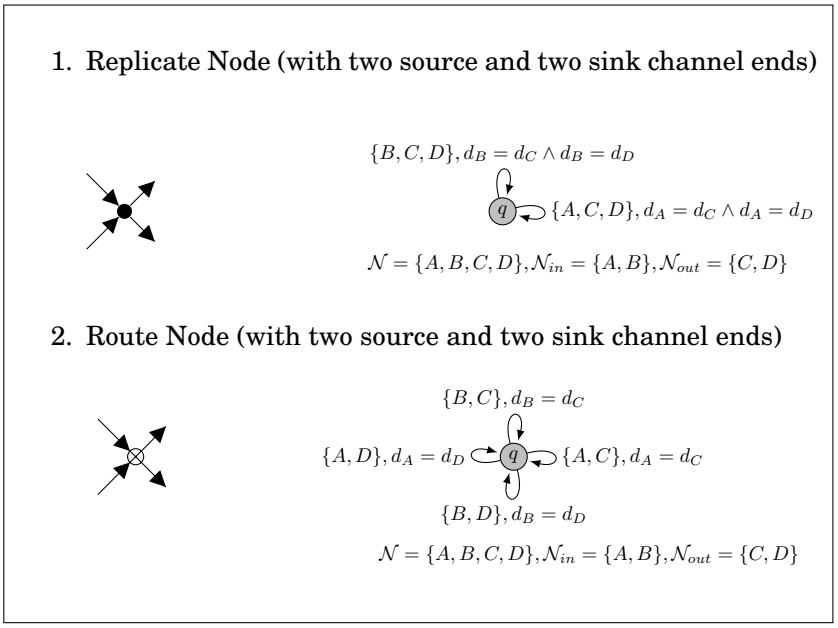


Figure 4.2: Reo route and replicate nodes.

channel ends, and a *mixed node* if it contains both source and sink channel ends.

In a Reo circuit, a source node or sink node can be seen as a point where data can be, respectively, written to or read from, from the outside world, and can thus serve as interface points, if that Reo circuit is to represent a component or a channel.

A node with replicate semantic is active with a value $d \in \mathfrak{D}$ if exactly one of its sink channel ends (or none in case of source nodes) as well as all its source channel ends (or none in the case of sink nodes), are all active with value d. On the other hand, a node with route semantics is active with a value $d \in \mathfrak{D}$ if exactly one of its sink channel ends (or none in case of source nodes) as well as exactly one of its source channel ends (or none in case of sink nodes) are active with value d.

Intuitively, a replicate node, sends the value of a non-deterministically chosen sink channel end to all its source channel ends; while, a route node, sends the value of a non-deterministically chosen sink channel end to a non-deterministically chosen source channel end.

Therefore, nodes don't simply have a single constraint automata for their semantics, rather, they are templates for constructing one, depending on the number of sink and source channel ends they have. Figure 4.2 depicts constraint automata corresponding to a replicate node and a route node each having two source channel ends and two sink channel ends.

4.1.3 Reo Circuits

A Reo circuit is simply a collection of components, channels (from library or user-defined) whose channel ends can be combined into nodes.

Figure 4.3, depicts an example of a Reo circuit. This circuit represents a synchronization mechanism. This system accepts input on its three source nodes, A,B and C, and stores their values into FIFO channels. The values received from nodes A, B and C (which are

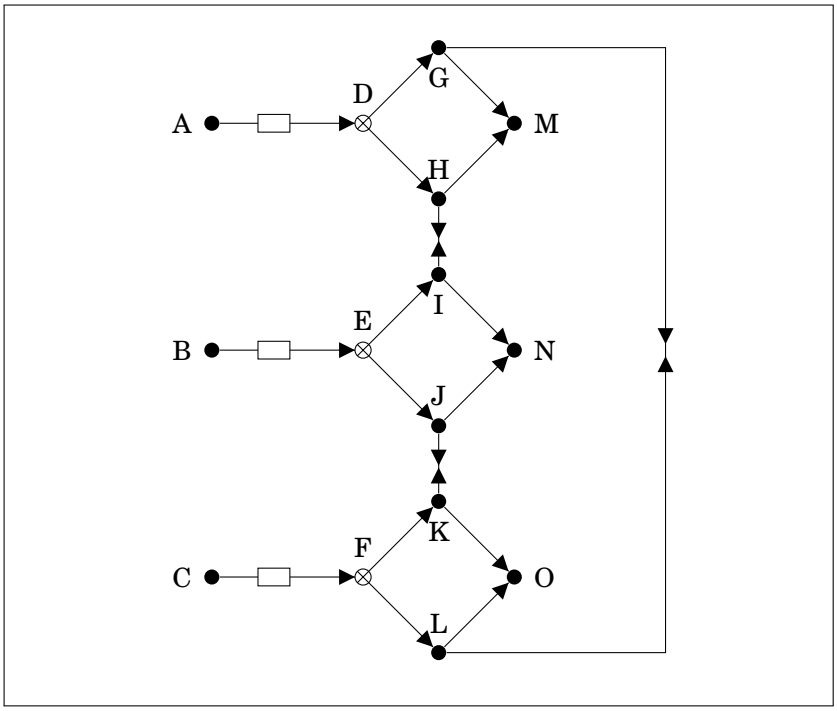


Figure 4.3: An example of a Reo Circuit.

now stored in FIFO channels), on the other hand, are respectively sent to sink nodes M, N and O, once there can be two of them sent simultaneously. In case all three values are ready, two of them are chosen non-deterministically. In this Reo circuit, nodes D, E and F are mixed nodes with route semantics while nodes G, H, I, J, K and L are mixed nodes with replicate semantics.

4.2 Vereofy

In this section, we are going to present an overall description of Vereofy (see [BKK11, BBKK09b, BKKa, BKKb, Klü12]), the model checking tool which uses constraint automata as its underlying model specification approach, developed at technical university of Dresden. It is equipped with the two languages CARML (Constraint Automata Reactive Module Language) and RSL (Reo Scripting Language). Both these languages are in fact equally powerful as they both specify constraint automata. While the former specifies constraint automata in an almost direct way by specifying state space of the automaton as well as its transitions, atomic propositions and so on, the latter provides the facility to specify a script (executable code) to construct Reo circuits where components can be expressed in CARML or be Reo circuits themselves.

Vereofy internally uses symbolic representation of constraint automata. It also is capable of verifying linear and branching time properties as well as checking bisimulation between two constraint automata. Although, bisimulation checking done by Vereofy is not exactly the same as the one we have provided in this thesis and only considers data-flow

of the constraint automata – it ignores atomic propositions. Please note that the notion of bisimulation presented in this thesis was to help show that the result of reduction can be effectively used to check certain temporal properties as a replacement for the original system.

We have implemented and integrated, as we will later see, some parts of the approach of symmetry reduction that was introduced in this thesis.

4.2.1 CARML (Constraint Automata Reactive Module Language)

CARML is a language – a variant of guarded command language (see [Dij78]) – which facilitates a low-level expression of constraint automata. This is done via providing a syntax to specify ports (data-flow locations), a number of variables called state variables, atomic propositions, and transitions which are used to specify CARML modules. Each of the variables comes together with a type as well as an initial value. Therefore, before introducing any feature of the CARML language we first introduce the data types.

Here, while we will describe various notions of CARML language, present examples with the agenda of putting them together at the end to form a CARML module corresponding to FIFO1 – A module very similar to the definition of builtin channel FIFO1 in vereofy – see [BKKa].

4.2.1.1 Data Types

In CARML modeling language we have data types for state variables and ports. These data types can be any of the following different types:

Integer Range An integer range with a lower bound 'l' and upper bound 'u', both inclusive, written as follows:

```
int(l, u)
```

The "`Bool`" data type is a specific case of integer range, i.e., "`int(0,1)`".

Enumeration Enumerations represent an enumeration of identifiers. As an example:

```
enum {id_1, id_2}
```

A variable v of the type of the enumeration above can have a value of either "`id_1`" or "`id_2`".

Struct Structures represent tuples of data types. Each element has its own data type. An example of a structure is as follows:

```
struct {type1 :  a; type2 :  b;}
```

The values can be accessed using the name of the element, e.g. `v.a==1`.

Tagged union A tagged union is a disjoint union of a number of data types. A variable with data type of a tagged union can at any time hold the value with the data type of one of its member data types. An example of a tagged union is as follows:

```
tagged union {type1 :  a; type1 :  b;}
```

In the example above, the value can be either of type type "`type1`" or type "`type2`". To check the type of the current value, a special member name called 'tag' is provided, e.g., "`v.tag==type1`" where `v` is a variable of the type of the tagged union above.

4.2.1.2 Ports (Data-Flow Locations)

In CARML, ports are specified directly by their name and direction (input or output). The lines below create an input port named 'A' and an output port 'B':

```
in:  A;
out:  B;
```

The values that can be transferred through ports (data domain), is the members of the data type 'Data'. This specific type can be specified explicitly in the input file where the system is designed or it can be given to vereofy model checking tool via a specific parameter. In case this type is not explicitly specified by the user the type 'int(0,1)' which can take up values '0' and '1' will automatically be assumed.

4.2.1.3 State Variables

The state space of the constraint automaton is specified using a number of variables each of which have a type (which is explained later on) and possibly an initial value. The following lines describe two variables: `X` of type `int(0,1)` without any initial value and a a variable `Y` of the type 'enum{empty, full}' with an initial value 'empty'.

```
var:  int(0,1) X;
var:  enum{empty, full} Y := empty;
```

The state space of the generated constraint automaton is the tuples of values of state variables. For instance if a CARML module has variable definitions as above, the set of states of the resulting constraint automata would be the set $\{(0, \texttt{empty}), (0, \texttt{full}), (1, \texttt{empty}), (1, \texttt{full})\}$ and the initial state space would be $\{(0, \texttt{empty}), (1, \texttt{empty})\}$.

4.2.1.4 Atomic Propositions

Given the provided facility of specifying state space of the constraint automaton as a number of variables, the definition of atomic propositions can be as easy as specifying conditions on state variables. As an example, given the state variables `X` and `Y` in the previous part, we can simply define atomic propositions to capture the states where the constraint automaton is in a state where `X` is empty as follows:

```
ap:  EMPTY <=> Y == empty;
```

4.2.1.5 Transitions

In CARML, transitions consist of three parts, state guards, IO guards and state assignments. State guards, are simply conditions that specify the states from which the transition can take place, IO guards on the other hand, are the counterparts for data constraints while the state assignments describe the target of the transition. As an example consider the following transitions for which we have assumed existence of the ports and variables as examples brought above.

```
Y == empty -[{A}]-> Y := full & X := #A;
Y == full -[{B} & #B == X]-> Y := empty;
```

Here, the first transition, can be used when the constrain automaton is in a state where the value of `X` is '`empty`' (state guard), and port `A` is the only port that is active (IO guard) and after the execution of the transition, the constraint automata goes to a state where value of `X` is '`full`' and the value of `Y` is the same as the value of port `A`. The second transition, on the other hand, can be used when the constrain automaton is in a state where the value of `X` is '`full`' (state guard), and port `B` is the only port that is active and its value is the same as the state variable `Y` (IO guard) and after the execution of the transition, the constraint automata goes to a state where value of `X` is '`empty`'. Please note that here the type Data (the data type designated for ports) is assumed to be the default type `int(0,1)`.

Even in this small example, one can easily see the conciseness of this approach. We can easily see that the transitions above are in fact describing the following transitions:

First Transition:

$(0,\texttt{empty}) \xrightarrow{\{A\},d_A=0} (0,\texttt{full})$

$(0,\texttt{empty}) \xrightarrow{\{A\},d_A=1} (1,\texttt{full})$

$(1,\texttt{empty}) \xrightarrow{\{A\},d_A=0} (0,\texttt{full})$

$(1,\texttt{empty}) \xrightarrow{\{A\},d_A=1} (1,\texttt{full})$

Second Transition:

$(0,\texttt{full}) \xrightarrow{\{A\},d_A=0} (0,\texttt{empty})$

$(1,\texttt{full}) \xrightarrow{\{A\},d_A=1} (1,\texttt{empty})$

Altogether, the CARML codes of examples above put together form a CARML module that resembles a first in first out queue (the FIFO1 channel). The following code is the code of such a module.

```
MODULE fifo {
    in:  A;
    out:  B;

    var:  int(0,1) X;
    var:  enum{empty, full} Y := empty;

    ap:  EMPTY <=> Y == empty;

    Y == empty -[{A}]-> Y := full & X := #A;
    Y == full -[{B} & #B == X]-> Y := empty;
}
```

4.2.2 RSL (Reo Scripting Language)

In spite of CARML which provides facility for a low level description of constraint automata, RSL provides a scripting language to construct Reo circuits. RSL provides facility for instantiation of CARML modules, other Reo circuits or built in channels which constructs an instance of them as a sub-component of the circuit being formed. It furthermore, provides facility to combine channel ends (and nodes) to form (or extend) nodes.

4.2.2.1 Sub-component Instantiation

In a Reo circuit, one can instantiate CARML modules, other Reo circuits or builtin circuits as sub-components of a circuit. As an example, consider the following Reo circuit:

```
CIRCUIT fifo3{
    f[0] = new fifo;
    f[1] = new fifo;
    f[2] = new fifo;
}
```

In this example, the Reo circuit 'fifo3' declares existence of three sub-components each of which are instances of the CARML module 'fifo' introduced earlier. In Reo circuits variables are created as they are assigned a value to and are all assumed to be arrays. Using the name of a variable without an array access, e.g., '`f`', is simply equivalent to accessing its first member (i.e., `f[0]`). In this example the variable '`f`' is an array of length 3 and each of the array cells contains a 'fifo' sub-component.

RSL, other than simple commands like '`new`' which is used to create sub-components, has the facility of loops and conditional statements. For example the circuit above could as well be created as shown below.

```
CIRCUIT fifo3{
    for(i=0;i<3;i=i+1){
        f[i] = new fifo;
    }
}
```

4.2.2.2 Channel Ends

In a Reo circuit, described in RSL, for each sub-component, there is a number of channel ends which represent the ports (if the sub-component is CARML module) or exported nodes (if the sub-component is another Reo circuit) of that sub-component. A channel end is of type 'source' if it represents an input port of a CARML module sub-component or a source node (a node which it self is collection of source channel ends) of a Reo circuit sub-component; and it is of type 'sink' if it represents an output port of a CARML module sub-component or a sink node (a node which it self is collection of sink channel ends) of a Reo circuit sub-component. In the example above, the channel ends of the three subcomponents can be accessed under the names:

```
f[0].A
f[0].B
f[1].A
f[1].B
f[2].A
f[2].B
```

Or equivalently (respectively):

```
f[0].source[0]
f[0].sink[0]
f[1].source[0]
f[1].sink[0]
f[2].source[0]
f[2].sink[0]
```

The arrays source and sink are the arrays of source channel ends and sink channel ends of each sub-component with which the circuit can access source and sink channel ends of each sub-component without needing to directly access them by their individual (local) names. Sizes of these arrays can respectively be accessed by '.sources' and '.sinks' for each sub-component[1].

4.2.2.3 Nodes

In RSL, nodes can be explicitly created or created via the key command 'join'. This command given two parameters joins them together according to their types. If both arguments are channel ends, a new replicate node is created with these two channel ends and the node is returned. If one argument is a channel end and the other is node, the channel end is simply attached to the node and the node (the argument) is returned. In this case, the operation actively alters the node argument and if the return value is assigned to a variable, the variable name that the return value was assigned to is simply considered as an alias for the node argument. If both arguments are nodes of the same semantics (replicate or route) then both nodes are considered equal to (overwritten) a node with the common semantics and the union of channel ends of both nodes, the node (which is equal to any of the arguments now) is returned. This operation also alters both nodes and causing them to be aliases of the same node. If the return value is assigned to a variable, the variable that the result was assigned to is also considered as an alias of the two nodes. In case the arguments of join operation are nodes with different semantics, the operation will result in an error and consequently failure to construct the Reo circuit. If the result of the join operation is not assigned to a variable, it is considered anonymous and it can no longer be accessed, except for the two latter cases where the node argument(s) is (are) changed and is (are) now equal to the result. As an example we can put the sink channel end of the each fifo (except for the last fifo) and the source channel end of the next fifo in a node to make a cascading line of fifo queues acting as a fifo queue of a size 3.

```
CIRCUIT fifo3{
    for(i=0;i<3;i=i+1){
        f[i] = new fifo;
    }
    for(i=0;i<2;i=i+1){
        join(f[i].sink, f[i+1].source);
    }
}
```

[1]The facility of accessing the number of source channel ends and sink channel ends is only implemented in the version where experiments with symmetry are being carried out and are not available in the release version yet.

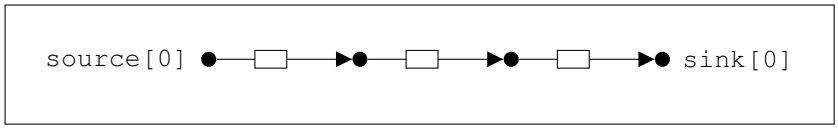


Figure 4.4: An example of a Reo Circuit for RSL Code.

4.2.2.4 Node Exporting and Hiding

In Vereofy, in a Reo circuit, nodes must be explicitly exported. For this purpose, there are two special arrays source and sink in each circuit (which we earlier used to access channel ends without referring to their local names). Each node that is to be exported should be assigned to a member of these arrays. It should although be noted that, as we explained earlier for Reo circuits, only source and sink nodes can be exported and mixed nodes are always considered internal and are not exportable. Furthermore, any node that is not exported is hidden from the constraint automaton of the result of the circuit. This hiding always happens except for the case where the circuit is considered the main circuit representing the system at hand. As an example, the source channel end of first fifo and the sink channel end of the last fifo in the last example can be exported (by joining – or assigning them, which is automatically assumed as joining by Vereofy – them with a member of source or sink array), so that the circuit would practically have a single source and a single sink node exported which would play the role of input and out for our 3 cell fifo queue, respectively. In this example, the mixed nodes, namely those anonymous nodes that are providing the connection between the consecutive fifo queues are automatically hidden, if this circuit is not considered the main circuit of the model.

```
CIRCUIT fifo3{
    for(i=0;i<3;i=i+1){
        f[i] = new fifo;
    }
    for(i=0;i<2;i=i+1){
        join(f[i].source, f[i+1].sink);
    }
    source[0] = f[0].source;
    sink[0] = f[2].sink;
}
```

The Reo circuit '`fifo3`', defined above, is depicted in Figure 4.4.

4.2.2.5 Circuit Parameters

Parameters for circuits provide a form of polymorphism for Reo circuits. In definition of circuits, there can be some parameters defined. In the body of circuits, parameters are dealt with as constant entities (e.g., numbers, name of CARML modules, name of Reo circuits, etc.). As an example consider the following circuit which generalizes fifo as defined earlier by getting the number of cells of the queue as a parameter.

```
CIRCUIT gen_fifo<k>{
    if(k==0){
        f = new SYNC;
        source[0] = f[0].source;
        sink[0] = f[0].sink;
    }
    if(k>=1){
        for(i=0;i<k;i=i+1){
            f[i] = new fifo;
        }
        for(i=0;i<k-1;i=i+1){
            join(f[i].source, f[i+1].sink);
        }
        source[0] = f[0].source;
        sink[0] = f[k-1].sink;
    }
}
```

Here, the name 'SYNC' refers to the builtin synchronous channel. In case a fifo queue of size zero is requested, the circuit simply produces a synchronizer and exports is ports so that the resulting constraint automaton would behave as a synchronizer which is expected of a queue of size zero – anything that is put into the queue should instantaneously be taken out from it.

5 Symmetry in Constraint Automata

Using symmetry in models to tackle the state space explosion problem is a well known approach which was widely used in various cases, e.g. [CEFJ96, CJ95, GS05, ID96, CEJS98, ES97, Ios02, MDC06, SG04, EW03, ET03, DM05, DMP07, DM06, KNP06, BDE+12], where groups of symmetry are used to define and reason about symmetry in models.

In this chapter, we are going to present a similar notion of symmetry specifically introduced for constraint automata. We will later discuss the complexity of certifying such symmetries and, most importantly, show that reducing a system with respect to such symmetry, results in a bisimilar system – if we only consider those atomic propositions that are preserved under the action of the symmetry group in question.

The notion of symmetry introduced here is similar to the one introduced in [CEFJ96], adapted to take into account constraints between states rather than their immediate reachability. There, they show that the result of symmetry reduction preserves satisfiability of CTL* formulas (by induction over the structure of CTL* formulas) that only use atomic propositions that are preserved by the group of symmetry at hand. Here, on the other hand, we show an equally strong result by showing that result of reduction is bisimilar to the original constraint automaton if only those atomic propositions are considered that are preserved by the symmetry group at hand. This is due to the fact that bisimulation and CTL* preservation are equivalent which was shown in [BCG88].

5.1 Symmetry for Contraint Automata

When dealing with symmetries in different contexts, it is usually the case that a restriction is put on permutations so that they capture a specific form of symmetry relevant to the context in question. Groups of symmetry that capture that form of symmetry are then defined to be groups acting on the structure in question while their permutations are of the specific form defined to capture that form of symmetry. As we discussed earlier, we are going to define symmetries in such a form that the result of reduction based on those symmetries shall be bisimilar to original automaton.

Definition 5.1 (Permutations for Constraint Automata)**.** Let $\mathcal{A}$ be a constraint automaton with the set of states Q and the set of data-flow locations $\mathcal{N}$. Then, a symmetry permutation $\sigma : Q \to Q$ for constraint automaton $\mathcal{A}$ is a bijective function such that:

$$\forall q_1, q_2 \in Q \; \forall N \subseteq \mathcal{N}. \left(\mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N) \equiv \mathfrak{C}^{\mathcal{A}}_{\sigma(q_1),\sigma(q_2)}(N) \right)$$

■

Before defining groups of symmetry for constraint automata, we show that if two bijective functions over the set of states of a constraint automaton are symmetry permutations for that constraint automaton, then their composition as well as their inverses are also symmetry permutations for that constraint automaton.

Theorem 5.2 (Composition and Inversion of Symmetry Permutations for Constraint Automata)**.** Let $\mathcal{A}$ be a constraint automata with the set of states Q and $\sigma_1 : Q \to Q$ and $\sigma_2 : Q \to Q$ be two symmetry permutations for $\mathcal{A}$. Then, $\sigma_1 \circ \sigma_2$, $\sigma_2 \circ \sigma_1$, σ_1^{-1} and σ_2^{-1} are all symmetry permutations for $\mathcal{A}$. ■

Proof. Here, we only show it for $\sigma_1 \circ \sigma_2$ and σ_1^{-1}, the proof for $\sigma_2 \circ \sigma_1$ and σ_2^{-1} are respectively similar and are thus left out.
To show $\sigma_1 \circ \sigma_2$, we only need to consider the fact that σ_1 and σ_2 are both symmetry permutations. From this, we know:

$$\forall q_1, q_2 \in Q \; \forall N \subseteq \mathcal{N}. \left(\mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N) \equiv \mathfrak{C}^{\mathcal{A}}_{\sigma_1(q_1),\sigma_2(q_2)}(N) \right) \tag{5.1}$$

and

$$\forall q_1, q_2 \in Q \; \forall N \subseteq \mathcal{N}. \left(\mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N) \equiv \mathfrak{C}^{\mathcal{A}}_{\sigma_2(q_1),\sigma_2(q_2)}(N) \right) \tag{5.2}$$

On the other hand, since σ_1 is a bijection, it maps each member of Q to some other member of it. Thus, according to (5.1), we can simply replace $\mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N)$ with $\mathfrak{C}^{\mathcal{A}}_{\sigma_1(q_1),\sigma_1(q_2)}(N)$ and $\mathfrak{C}^{\mathcal{A}}_{\sigma_2(q_1),\sigma_2(q_2)}(N)$ with $\mathfrak{C}^{\mathcal{A}}_{\sigma_1(\sigma_2(q_1)),\sigma_1(\sigma_2(q_2))}(N)$ in Formula (5.2), which would result in:

$$\forall q_1, q_2 \in Q \; \forall N \subseteq \mathcal{N}. \left(\mathfrak{C}^{\mathcal{A}}_{\sigma_1(q_1),\sigma_1(q_2)}(N) \equiv \mathfrak{C}^{\mathcal{A}}_{\sigma_1(\sigma_2(q_1)),\sigma_1(\sigma_2(q_2))}(N) \right) \tag{5.3}$$

Furthermore, according to (5.1) we can simply replace $\mathfrak{C}^{\mathcal{A}}_{\sigma_1(q_1),\sigma_1(q_2)}(N)$ with $\mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N)$, in (5.3), since they are equal, which results in:

$$\forall q_1, q_2 \in Q. \left(\mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N) \equiv \mathfrak{C}^{\mathcal{A}}_{\sigma_1(\sigma_2(q_1)),\sigma_1(\sigma_2(q_2))}(N) \right)$$

Which means $\sigma_1 \circ \sigma_2$ is a symmetry permutation for $\mathcal{A}$.
On the other hand, to show σ_1^{-1} is also a symmetry permutation, since σ_1^{-1} is itself a bijection over Q and according to (5.1), we can simply replace $\mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N)$ with $\mathfrak{C}^{\mathcal{A}}_{\sigma_1^{-1}(q_1),\sigma_1^{-1}(q_2)}(N)$ and replace $\mathfrak{C}^{\mathcal{A}}_{\sigma_1(q_1),\sigma_1(q_2)}(N)$ with $\mathfrak{C}^{\mathcal{A}}_{\sigma_1(\sigma_1^{-1}(q_1)),\sigma_1(\sigma_1^{-1}(q_2))}(N)$ in (5.1) itself and get:

$$\forall q_1, q_2 \in Q \; \forall N \subseteq \mathcal{N}. \left(\mathfrak{C}^{\mathcal{A}}_{\sigma_1^{-1}(q_1),\sigma_1^{-1}(q_2)}(N) \equiv \mathfrak{C}^{\mathcal{A}}_{\sigma_1(\sigma_1^{-1}(q_1)),\sigma_1(\sigma_1^{-1}(q_2))}(N) \right)$$

Which, according to group theoretic notions, can be simplified to

$$\forall q_1, q_2 \in Q \; \forall N \subseteq \mathcal{N}. \left(\mathfrak{C}^{\mathcal{A}}_{\sigma_1^{-1}(q_1),\sigma_1^{-1}(q_2)}(N) \equiv \mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N) \right)$$

Which means σ_1^{-1} is a symmetry permutation for $\mathcal{A}$. □

Hence, we can use symmetry permutations for constraint automata to form symmetry groups for them.

Definition 5.3 (Groups acting on Constraint Automata)**.** Let $\mathcal{A}$ be a constraint automaton. Then, a group acting on $\mathcal{A}$ is a group $\mathcal{G} = \langle G, \circ, .^{-1}, id \rangle$ where G is a set of permutations for constraint automaton $\mathcal{A}$.

■

Remark 5.4 (Groups Acting on Constraint Automata versus groups Acting on State Set of Constraint Automata)**.** This is very important to note that not any group that is acting on the set of states of a constraint automaton is a symmetry group of that constraint automaton (acting on the automaton). Hence, whenever we say a group is acting on the set of states of a constraint automaton it does not necessarily mean that it is acting on the constraint automaton itself. ■

According to Theorem 5.2, we can easily see that a group that is represented by its generating set is acting on a constraint automaton if and only if all of the members of the generating set are symmetry permutations for that constraint automata, since all members are generated from members of generating set by applying function composition and inversion.

5.1.1 Complexity of Certifying Symmetry

Similarly to the problem of bismilarity, the problem of checking whether a given bijective function over the set of states of a constraint automaton, is a symmetry permutation for that constraint automaton, Definition 5.5, is **CoNP**-Complete. The proof provided here for hardness is very similar to the proof of **CoNP**-hardness of bisimulation problem for constraint automata in [BSAR06]. For further reading on computational complexity and the propositional validity problem, please refer to [AB09].

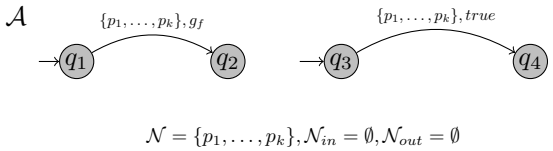
Definition 5.5 (Symmetry Permutation Certification Problem for Constraint Automata)**.** Given a constraint automaton, $\mathcal{A} = \langle Q, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out}, \rightarrow, Q_0 \rangle$, a bijective function $\sigma : Q \rightarrow Q$ and a data domain $\mathfrak{D}$, the symmetry permutation certification problem is the decision problem whether σ is a symmetry permutation for $\mathcal{A}$ considering data domain $\mathfrak{D}$. ■

Theorem 5.6 (**CoNP**-Completeness of Certification of Symmetry Permutation for Constraint Automata)**.** Given a constraint automaton $\mathcal{A}$ with the set of states Q and a bijective function $\sigma : Q \rightarrow Q$, assuming data domain $\mathfrak{D}$, the problem of checking whether σ is a symmetry permutation for $\mathcal{A}$ is **CoNP**-Complete. ■

Proof. To show **CoNP**-Hardness, we do a reduction from propositional validity problem – which is well known to be **CoNP**-Complete – to the problem of certification of symmetry permutation – the reduction is very similar to the proof of **CoNP**-Hardness of bisimilarity for constraint automaton, presented in [BSAR06]. To do so, we simply consider a propositional formula f which only uses propositional variables $p_1, \ldots, p_k$. It is easy to see that formula f is a valid propositional formula if and only if σ such that

$$\sigma(q_1) = q_3,\ \sigma(q_2) = q_4,\ \sigma(q_3) = q_3,\ \sigma(q_4) = q_4$$

is a symmetry permutation for the following constraint automata, $\mathcal{A}$, assuming $\mathfrak{D} = \{0, 1\}$.



To show **CoNP**-Containment, on the other hand, we simply consider the complementary problem – the problem where the given bijective function is not a symmetry permutation for the provided constraint automaton – and show that it can be solved with a non-deterministic polynomially time-bounded turing machine. In order to show that a given permutation σ is not a symmetry permutation for a constraint automaton $\mathcal{A}$ with the set of states Q and set of data-flow locations $\mathcal{N}$ for some data domain $\mathfrak{D}$, we can non-deterministically choose a set of data-flow locations $N \subseteq \mathcal{N}$, a CIO $c \in \mathfrak{Cio}(\mathcal{N})$ and two states q and q' simply check if

$$c \models \mathfrak{C}^{\mathcal{A}}_{q,q'}(N) \text{ and } c \not\models \mathfrak{C}^{\mathcal{A}}_{\sigma(q),\sigma(q')}(N)$$

or

$$c \not\models \mathfrak{C}^{\mathcal{A}}_{q,q'}(N) \text{ and } c \models \mathfrak{C}^{\mathcal{A}}_{\sigma(q),\sigma(q')}(N)$$

holds. Which can be simply carried out by a deterministic polynomial time-bounded turing machine. □

The complexity results above are pertaining to symmetry permutations. Yet, we can easily see that these results can be extended to the case of symmetry groups as well. In other words, certification of symmetry groups for constraint automata is also **CoNP**-Complete.

We can simply see that to certify a group's action on a constraint automaton, we simply need to check all its permutations (or equivalently permutations of its generating set). Which means carrying out a **CoNP**-Complete task a number of times polynomially (in fact linearly) bound to the size of the group at hand. Which is trivially **CoNP**-Complete.

5.2 Reduction and Symmetry

So far, we have introduced the notion of symmetry and groups of symmetry for constraint automata. We introduced these notions in order to use them for reducing constraint automata in presence of a symmetry in the system. In order to do so, we will first introduce symmetry reduction for constraint automata. Later, we will show that such reductions would result in constraint automata that are bisimilar to the original automata, if only those atomic propositions are considered that are preserved by the symmetry group. More precisely, we show that if we only consider those atomic propositions that the group is invariant for the resulting constraint automaton is bisimilar to the original constraint automaton, considering only those atomic propositions that the group in question is an invariance group for. A group is said to be an invariance group for a set of atomic propositions if for any orbit of that symmetry group all states are labeled with the same subset of that set of atomic propositions.

5.2.1 Symmetry Reduction

Given a constraint automaton and a group acting on it, the result of reduction of that constraint automaton with respect to the given group symmetry is a constraint automaton where each state is substituted with its orbit and for the transitions and labels of the states of the reduced constraint automaton (orbits) are obtained from an arbitrary but fixed member of that orbit.

Definition 5.7 (Symmetry Reduction for Constraint Automata)**.** Given constraint automaton $\mathcal{A} = \langle Q, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out} \rightarrow, Q_0, AP, L \rangle$, a group $\mathcal{G}$ that acts on it, the result of reducing $\mathcal{A}$

with respect to $\mathcal{G}$, denoted by $\mathcal{A}_{/\mathcal{G}}$, is defined as follows:

$$\mathcal{A}_{/\mathcal{G}} = \langle R, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out} \rightarrow', R_0, AP, L' \rangle$$

such that,

- $R = \{\Theta_{\mathcal{G}}(q) \mid q \in Q\}$
- $R_0 = \{\Theta_{\mathcal{G}}(q) \mid \exists q \in Q_0\}$
- $\rightarrow' = \{(r_1, N, g, r_2) \mid r_1, r_2 \in R \wedge rep(r_1) \xrightarrow{N,g} rep(r_2)\}$
- $L'(r) = L(rep(r))$

where, $rep : 2^Q \rightarrow Q$ is a function that maps each set of states to a unique representative (an arbitrary but fixed member) of that set. Furthermore, in the sequel, we consider $\mathfrak{Red}_{\mathcal{G}} = \{(q, \Theta_{\mathcal{G}}(q)) \mid q \in Q\}$ as the reduction relation for symmetry group $\mathcal{G}$, where Q is the set that $\mathcal{G}$ is acting on.

■

Here, we can see that each orbit gets the labels of a single (chosen by rep) member of itself. Therefore, there can possibly be a state $q' \in \Theta_{\mathcal{G}}(q)$ for which $L(q') \neq (L(\Theta_{\mathcal{G}}(q)) = L'(\Theta_{\mathcal{G}}(q)))$. Therefore, here, we define the notion of groups being invariant for some set of atomic propositions, which captures those atomic propositions for which we always have $L(q) = L(rep(\Theta_{\mathcal{G}}(q))) = L'(\Theta_{\mathcal{G}}(q))$.

5.2.2 Group invariance for Atomic Propositions and AP-Restricted Constraint Automata

A group $\mathcal{G}$ is said to be invariant for a set of atomic propositions B if for any atomic proposition in $b \in B$, members of each of the orbits of $\mathcal{G}$, either all have b as one of their labels or they all don't.

Definition 5.8 (Label Invariance)**.** Let $\mathcal{A} = \langle Q, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out} \rightarrow, Q_0, AP, L \rangle$ be constraint automaton, $\mathcal{G}$ be a a group acting on $\mathcal{A}$ and $AP' \subseteq AP$ be a set of atomic propositions. Then, we say $\mathcal{G}$ is an invariance group of AP' if and only if:

$$\forall_{q \in Q} \forall_{q' \in \Theta_{\mathcal{G}}(q)} : L(q) \cap AP' = L(q') \cap AP'$$

■

On the other hand, since we are going to establish a relation between the original constraint automaton and the result of reducing it with respect to some reduction relation justified by a group acting on the original constraint automaton, we need to have a way to restrict the atomic propositions of a constraint automata. We define this restriction as a constraint automaton that is identical to the original constraint automaton except for the set of atomic propositions and labeling function. The atomic proposition restriction restricts the labels of a constraint automaton by simply removing those that are not members of the restricted atomic propositions set.

Definition 5.9 (AP-Restricted Constraint Automata)**.** Given a constraint automaton $\mathcal{A} = \langle Q, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out} \rightarrow, Q_0, AP, L \rangle$ and a set B of atomic propositions the result of restricting $\mathcal{A}$ to B, denoted by $\mathcal{A}_{|_B}$, is defined as follows:

$$\mathcal{A}_{|_B} = \langle Q, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out} \rightarrow, Q_0, AP \cap B, L' \rangle$$

where

$$L'(q) = L(q) \cap B$$

■

5.3 Bismilarity of the Result of Reduction with Respect to Symmetry

Here, we show that given a constraint automaton $\mathcal{A}$, a group $\mathcal{G}$ acting on it such that $\mathcal{G}$ is invariant of a set of atomic propositions B and the result of reducing $\mathcal{A}$ with respect to $\mathcal{G}$, we have, $\mathcal{A}_{|_B}$ and $(\mathcal{A}_{/\mathcal{G}})_{|_B}$ are bisimilar.

Theorem 5.10 (Bisimilarity of Symmetry Reduction)**.** Let $\mathcal{A}$ be a constraint automaton with the set of states Q and $\mathcal{G}$ be a group acting on it that is invariant of B, a subset of atomic propositions of $\mathcal{A}$. Then, $\mathfrak{Red}_{\mathcal{G}}$ is a bisimulation relation for $(\mathcal{A}_{/\mathcal{G}})_{|_B}$ and $\mathcal{A}_{|_B}$. ■

Proof. Let $\mathcal{A}_{|_B} = \langle Q, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out}, \rightarrow, Q_0, B, L_1 \rangle$ be the original constraint automaton restricted to B and $(\mathcal{A}_{/\mathcal{G}})_{|_B} = \langle R, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out}, \rightarrow', R_0, B, L_2 \rangle$ be the result of restricting $\mathcal{A}_{/\mathcal{G}}$ to B.

According to the definition of reduction we have, $R_0 = \{\Theta(q) \mid q \in Q_0\}$. Therefore, we have:

$$\forall q \in Q_0 \exists r \in R_0.\ (q, r) \in \mathfrak{Red}_{\mathcal{G}}$$

and

$$\forall r \in R_0 \exists q \in Q_0.\ (q, r) \in \mathfrak{Red}_{\mathcal{G}}$$

In other words, for any initial state of $\mathcal{A}_{|_B}$, there is an initial state of $(\mathcal{A}_{/\mathcal{G}})_{|_B}$ which are related by $\mathfrak{Red}_{\mathcal{G}}$ and vice versa.
Since we have $\mathcal{G}$ is an invariance group of B, we have, any two states that are symmetric have the same atomic propositions (as those not complying have already been removed):

$$\forall q' \equiv_{\mathcal{G}} q.\ L_1(q') = L_1(q) \tag{5.4}$$

From Formula (5.4) and the fact that the representative of each set of states is in that set ($\forall Q' \subseteq Q.\ rep(Q') \in Q'$), we get:

$$\forall (q, r) \in \mathfrak{Red}_{\mathcal{G}}.\ L_1(q) = L_1(rep(r))$$

and consequently,

$$\forall (q, r) \in \mathfrak{Red}_{\mathcal{G}}.\ L_1(q) = L_2(r)$$

Which means any state q from $\mathcal{A}_{|_B}$ and any state r from $(\mathcal{A}_{/\mathcal{G}})_{|_B}$ that are related by $\mathfrak{Red}_{\mathcal{G}}$ have the same set of atomic propositions, that is condition (B.1) of Definition 3.13.

On the other hand, as $\mathcal{G}$ is a group acting on $\mathcal{A}$ and in turn also $\mathcal{A}_{|_B}$ – as they both have the same set of states and atomic propositions and the same transition relation –, we have:

$$\begin{aligned}
&(q_1, \Theta_{\mathcal{G}}(q_1)), (q_2, \Theta_{\mathcal{G}}(q_1)) \in \mathfrak{Red}_{\mathcal{G}} \Rightarrow \\
&\forall N \subseteq \mathcal{N}.\ \Big(\mathfrak{C}^{\mathcal{A}_{|_B}}_{q_1,q_2}(N) \equiv \mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N) \equiv \mathfrak{C}^{\mathcal{A}}_{q_1,rep(\Theta_{\mathcal{G}}(q_2))}(N) \equiv \mathfrak{C}^{\mathcal{A}}_{rep(\Theta_{\mathcal{G}}(q_1)),q_2}(N) \\
&\qquad\qquad \equiv \mathfrak{C}^{\mathcal{A}}_{rep(\Theta_{\mathcal{G}}(q_1)),rep(\Theta_{\mathcal{G}}(q_2))}(N) \equiv \mathfrak{C}^{\mathcal{A}_{/\mathcal{G}}}_{\Theta_{\mathcal{G}}(q_1),\Theta_{\mathcal{G}}(q_2)}(N) \\
&\qquad\qquad \equiv \mathfrak{C}^{(\mathcal{A}_{/\mathcal{G}})_{|_B}}_{\Theta_{\mathcal{G}}(q_1),\Theta_{\mathcal{G}}(q_2)}(N)\Big)
\end{aligned} \tag{5.5}$$

Which means, for any pair of states q_1 and q_2 from $\mathcal{A}_{|_B}$, the constraints between them is equivalent with the constriants between their representatives (orbits) in $(\mathcal{A}_{/\mathcal{G}})_{|_B}$.
As a result, for $(q_1, \Theta_{\mathcal{G}}(q_1)) \in \mathfrak{Red}_{\mathcal{G}}$; we can take $r_2 = \Theta_{\mathcal{G}}(q_2)$ as the other state in $(\mathcal{A}_{/\mathfrak{Red}})_{|_B}$ that is bisimilar to q_1. This choice together with Equation (5.5) result in the condition (B.2) of Definition 3.13 hold and be as follows:

$$\forall q_2 \in Q\ \exists r_2 \in R\ \forall N \subseteq \mathcal{N}.\ \Big(\Big(\mathfrak{C}^{\mathcal{A}_{|_B}}_{q_1,q_2}(N) \equiv \mathfrak{C}^{(\mathcal{A}_{/\mathfrak{Red}})_{|_B}}_{\Theta_{\mathcal{G}}(q_1),r_2}(N)\Big) \wedge (q_2, r_2) \in \mathfrak{Red}_{\mathcal{G}}\Big)$$

On the other hand, we can take $q_2 = rep(\Theta_{\mathcal{G}}(q_2))$ as the other state in $\mathcal{A}_{|_B}$ that is bisimilar to q_2. This choice together with Equation (5.5) and the fact that $\forall q \in Q.\ \Theta_{\mathcal{G}}(rep(\Theta_{\mathcal{G}}(q))) = \Theta_{\mathcal{G}}(q)$, result in the condition (B.3) of Definition 3.13 hold and be as follows:

$$\forall \Theta_{\mathcal{G}}(q_1) \in R\ \exists q_2 \in Q\ \forall N \subseteq \mathcal{N}.\ \Big(\Big(\mathfrak{C}^{\mathcal{A}_{|_B}}_{q_1,q_2}(N) \equiv \mathfrak{C}^{(\mathcal{A}_{/\mathfrak{Red}})_{|_B}}_{\Theta_{\mathcal{G}}(q_1),\Theta_{\mathcal{G}}(q_2)}(N)\Big) \wedge (q_2, \Theta_{\mathcal{G}}(q_2)) \in \mathfrak{Red}\Big)$$

□

6 Symmetry of Components

In Chapter 5, we defined the notion of symmetry and symmetry reduction for constraint automata. We, specifically, showed (Theorem 5.10) that constraint automata remain bisimilar – if only those atomic propositions are considered that group is invariant for (Definition 5.8) – under symmetry reduction.

These results show that, in practice, the notion of symmetry, as defined earlier, simply captures bisimilarities within a system which are usually rather hidden (by atomic propositions). The aspect of revealing hidden bisimilarities is a double-edged sword; while it can be useful to capture bisimilarities that wouldn't have been visible otherwise, not any kind of symmetry is useful as it may cause loss of information (labels of states) that are of crucial importance in verification of the property at hand. On the other hand, symmetry can be extended to the level of components, where the symmetry itself can be evident from the high-level layout of system design. These reasons render blind search for symmetries in a system improper and rather impractical. Although, Symmetry can be sought out in a system on the level of components automatically. Donaldson et. al. do this in [DM05], where they consider analysis of channel based diagrams of communications among components and use the information provided by those diagrams to detect symmetries.

As an instance, consider the case of a system with a number of identical components that process an input producing an output. Assume that the system sends the requests it received to a non-deterministically chosen idle component for it to be processed and gathers the results of components that have a result ready to the output. In this scenario, the identical components carrying out process of inputs are used symmetrically – inputs and outputs of them are handled symmetrically (non-deterministic choice). Therefore, we can simply forget about their order and for each state of the composite system simply consider the number of processor components that are idle and number of components that are not.

Therefore, in this chapter, we lift the notion of symmetry to the level of components of the system. Assuming that the designer of the system, alongside providing the model of the system, provides the symmetry of components of the system as well. The approach of obtaining symmetry over the states of the composite system from the symmetry expressed over components of the system has widely been used in various applications of symmetry to model checking, e.g., [CEFJ96, ET03, DMP07, DM06, EW03, GS05, CEJS98, BDE+12].

This lifting of the notion of symmetry then, allows us to inspect the symmetry provided by the system designer on the level of components of the system to confirm or reject the symmetry and do the symmetry reduction of the system without requiring to build and store the whole system beforehand.

6.1 Symmetry of Identical Components

As we discussed earlier, we can see the notion of symmetry of components as the equivalence of behavior of a system in which we keep track of state of every single component and a system in which we only keep track of number of components in each state (idle or working in the example above) – if the system is seen as a black box, as properties that depend on the order of those components will not be preserved. Ergo, we can use this intuition to take the symmetry on the level of components to the level of states of the composite system. States of the composite system are tuples of states of individual components. Hence, in each tuple we count the number of symmetric components in each state (all symmetric components are identical and thus have the same set of states). For instance in the example above we consider two states of the composite system symmetric if they agree on the state of non-symmetric components and the number of idle and working processors.

Taking this approach, however, as expected, would cause loss of some information of the system, in particular, those labels that rely on the order of symmetric components to be distinguished; as they no longer can be distinguished, as we only keep track of number of components in each state rather than the state individual components are in.

Moreover, we will see that in many cases, if not all, this loss of information of the model is not restricted to the labels and also includes the information on data-flow to and from individual components for which there are other components that are symmetric to them. This is due to the fact that, as we are not keeping track of the states of individual components (those that have a symmetric counterpart in the system), we cannot know to which component the data-flow pertains. For further clarity, consider the aforementioned example of a system with identical processor components, if we are going to only consider the number of processors that are idle or working, we cannot know to which idle processor the received request is being assigned – all we know is that it is being assigned to some idle processor. That is why, as we will see, that data-flow locations of components with symmetric counterparts are in many cases blocking the symmetry as the result of translating component level symmetry to the level of states of the components would not hold unless if they are hidden. We will later elaborate on this phenomenon.

In this chapter, we put the restriction that the designer of the system can only declare two component symmetric, if they are identical – they only differ in the name of data-flow locations. Under this assumption, we can take the symmetry of components down to the level of states of the composite system by abstracting away the order of states of symmetric components. Therefore, given an instance of component symmetry, we can take that symmetry to the level of states of the composite system and check its validity and carry out the reduction on the level of states of the composite system without requiring to build and store the whole composite system.

6.1.1 Identical Components

As we discussed earlier, we are going to allow two components to be declared symmetric by the system designer if the components are identical. Two components are considered identical if they only differ in the name of their data-flow locations. In other words, two components are considered symmetric if renaming data-flow locations of one could result in the other.

Before, formally defining this notion, we first define the notion of data-flow mapping. That is, using a bijection to map data-flow location of a constraint automaton. The result of data-flow mapping a constraint automaton $\mathcal{A}$ with a set of data-flow locations $\mathcal{N}$ with

respect to a bijective mapping over its data-flow locations $f : \mathcal{N} \to \mathcal{N}'$ is denoted by $\mathcal{A}_{\mathcal{N} \to_f \mathcal{N}'}$ and is defined as follows:

Definition 6.1 (Data-Flow Mapping)**.** Let $\mathcal{A} = \langle Q, \mathcal{N}, \mathcal{N}_{in}, \mathcal{N}_{out}, \to, Q_0, AP, L \rangle$ be a constraint automaton and $f : \mathcal{N} \to \mathcal{N}'$ be a bijective mapping. Then, the result of mapping data-flow locations of $\mathcal{A}$ with respect to f, $\mathcal{A}_{\overrightarrow{f}}$, is as follows:

$$\mathcal{A}_{\overrightarrow{f}} = \langle Q, \mathcal{N}', \mathcal{N}'_{in}, \mathcal{N}'_{out}, \to', Q_0, AP, L \rangle$$

where,

$$\mathcal{N}'_{in} = \{f(A) \mid A \in \mathcal{N}_{in}\}$$
$$\mathcal{N}'_{out} = \{f(A) \mid A \in \mathcal{N}_{out}\}$$

and,

$$\frac{q_1 \xrightarrow{N,g} q_2}{q_1 \xrightarrow{N',g'} {}' q_2}$$

where,

$$N' = \{f(A) \mid A \in N\}$$

and g' is the result of substituting d_A by $d_{f(A)}$ in g. ■

Now, given the notion of data-flow mapping, we can simply define the notion of identical components by saying, two components (constraint automata) are identical if the result of chaining the data-flow locations of one of them with a data-flow mapping that maps its data-flow locations to the data-flow locations of the other results in the other constraint automaton.

Definition 6.2 (Identical Constraint Automata)**.** Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two constraint automata such that for $i \in \{1, 2\}$, we have:

$$\mathcal{A}_i = \langle Q, \mathcal{N}_i, \mathcal{N}_{i,in}, \mathcal{N}_{i,out}, \to_i, Q_0, AP, L \rangle$$

Then, $\mathcal{A}_1$ and $\mathcal{A}_2$ are identical if there is a bijective function $f : \mathcal{N}_1 \to \mathcal{N}_2$ such that:

$$\mathcal{A}_2 = \mathcal{A}_{1_{\overrightarrow{f}}}$$

■

6.1.2 Symmetry Translation

As we discussed earlier, according to the fact that we have put the restriction that only those identical components can be declared symmetric by system designer; we can translate this symmetry to the level of states of the composite system. To do so, we assume two states of the composite system are symmetric if they agree on the state of individual components that have no symmetric counter part in system and the number of symmetric components that are in each state (of the state space of the symmetric components).

In plain words, if the system is composed of n components $\mathcal{A}_1, \ldots, \mathcal{A}_n$ and $\mathcal{A}_k$ and $\mathcal{A}_l$ ($1 \leq k < l \leq n$) are symmetric (and identical) then a state $(q_1, \ldots, q_n)$ (a state where for $1 \leq i \leq n$ $\mathcal{A}_i$ is in state q_i) of the composite system, and another state of the composite system,

$$(q_1, \ldots, q_{k-1}, q_l, q_{k+1}, \ldots, q_{l-1}, q_k, q_{l+1}, \ldots, q_n)$$

should be declared symmetric in the result of translation.

In order to formally define this notion we assume that the designer of the system has provided us with the group acting on the components of the system they have specified. Therefore, we translate such a group into a group on the states of the composite system.

Definition 6.3 (Symmetry Translation)**.** Let $S = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ be a system consisting of n components (constraint automata) and $\mathcal{G} = \langle G, \circ, .^{-1} \rangle$ be a group acting on components of S such that components that are considered symmetric by $\mathcal{G}$ are identical. Then, the result of translation of $\mathcal{G}$ to the level of states of the composite system, is a group acting on the set of states of $\mathcal{A}_1 \bowtie \ldots \bowtie \mathcal{A}_n$, denoted by $\mathcal{G}_{\bowtie}$ and defined as follows:

$$\mathcal{G}_{\bowtie} = \langle \{\sigma_{k,l} \mid k \neq l \wedge \exists \sigma \in G.\ \sigma(\mathcal{A}_k) = \mathcal{A}_l\}, \circ, .^{-1} \rangle$$

where,

$$\sigma_{k,l}(q_1, \ldots, q_n) = (q_{i_1}, \ldots, q_{i_n})$$

such that,

$$q_{i_j} = \begin{cases} q_l & \text{if } i_j = k \\ q_k & \text{if } i_j = l \\ q_{i_j} & \text{otherwise} \end{cases}$$

■

Given the possibility to translate the symmetry expressed on the level of components of the system to the states of the composite system, we have grounds paved for formally defining symmetry for components. One naive way to do so would be to translate the given symmetry on the level of components to the symmetry on the level of states of the composite system and check whether the translated symmetry holds in the composite system. Yet, before doing so, one should pay attention to the problem of data-flow to and from components that have symmetric counterpart in the system.

6.1.3 Component Symmetry and Symmetry Blockage

A moment's reflection on the process of translation (Definition 6.3), should make it crystal clear that we are considering two states symmetric if they correspond to swapping states of symmetric components. Whereas, these states would have transitions to other symmetric states but with a different data-flow. For further clarity, let's turn back to the example of a system with processor components. That system would send each received request to an idle processor to be processed. If in one state of the system only the i-th processor is idle and a request comes, the system picks the i-th processor component to pass the request to. During the transition that passes the data through, the data-flow location between the broker part of the system and the i-th processor would be active with the request on the line. Whereas, considering another state which is symmetric to this state only to differ in the j-th ($j \neq i$) component being the only idle processor component. This time, the data-flow location between the part distributing the input and the j-th processor will be active carrying the request being passed. Therefore, even though, we expected these two state to be symmetric, their data-flow locations are blocking this symmetry. Example 6.4, provides a more detailed example of this phenomenon.

Example 6.4 (Symmetry Blockage)**.** Figure 6.1 shows the Reo circuit of a component together with the underlying constraint automaton for the component 'alt' (alternator). This

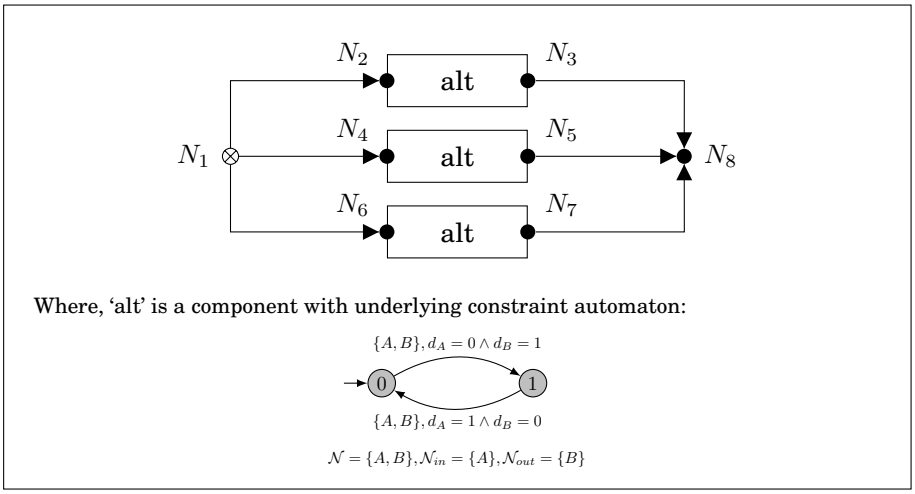


Figure 6.1: Example of component symmetry blocked by data-flow locations. See Example 6.4.

component simply alternates between accepting 'zero' on its source channel end (represented by data-flow location A) and sending 'one' out on its sink channel end (represented by data-flow location B) and accepting 'one' on its source channel and and sending 'zero' out on its sink channel end.

The circuit has a single source node (N_1) and a single sink channel end (N_8). This system non-deterministically chooses one of the 'alt' components that accept the value on its source node and sends that value to it, which causes the negation of that value to be sent on its sink node.

The constraint automaton of the composite system of this circuit is depicted in Figure 6.2. In this figure, nodes $N_1, \ldots, N_8$ are represented by A, A_1, B_1, A_2, B_2, A_3, B_3 and B, respectively. Moreover, the name of states only reflect the states of individual 'alt' components. This is due to the fact that the constraint automata corresponding to nodes have a single state and would just be repeated in each state just making the names bigger.

Assuming $\mathcal{G}$ is the symmetry group that declares all 'alt' components symmetric, we can easily see that the result of translating group $\mathcal{G}$ to the level of states of the composite system, $\mathcal{G}_{\bowtie}$, does not act on the system depicted in Figure 6.2.

Yet, $\mathcal{G}_{\bowtie}$ acts on this system after hiding data-flow locations A_1, B_1, A_2, B_2, A_3 and B_3, which correspond to nodes $N_2, \ldots N_7$, the result of which is depicted in Figure 6.3. The orbits of group $\mathcal{G}_{\bowtie}$ are depicted as translucent ovals in Figure 6.3. ■

Even though, symmetry can be blocked by data-flow locations of symmetric components, as shown in Example 6.4, this is not always the case. Consider a case similar to the system in Example 6.4, where instead of distributing input to the source channel end over the 'alt' components and sending their result out through the sink channel end of the system; the system would send an incoming value to all 'alt' components and require them to produce the same output in order for that output to be accepted and sent out through the systems sink channel end. In this case, even without hiding data-flow locations of the symmetric components, symmetry would be evident and that is due to the fact that all the source channel ends of all 'alt' components and all their channel ends would always have the same

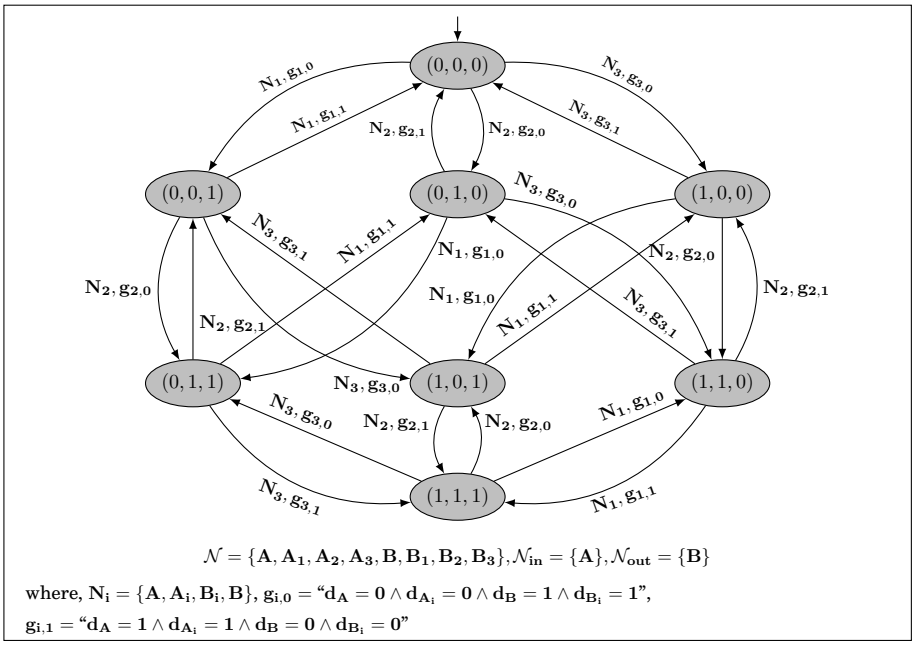


Figure 6.2: Example of component symmetry blocked by data-flow locations – composite system. See Example 6.4.

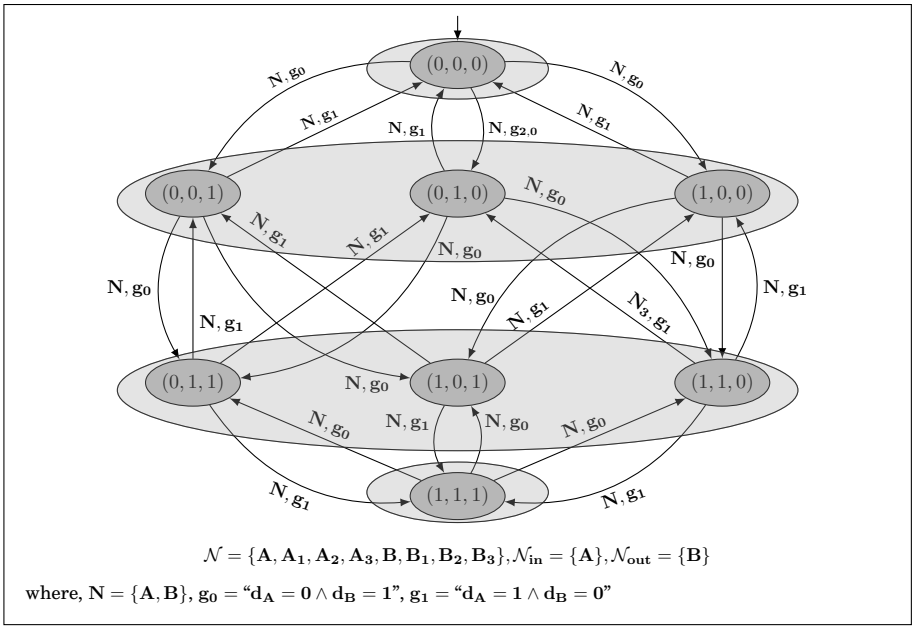


Figure 6.3: Example of component symmetry hidden by data-flow locations – composed system after blocking data-flow locations of symmetric components. See Example 6.4.

activation and valuation throughout all transitions of the system, which is a phenomenon that rarely occurs in real world situations.

Nevertheless, following the line of reasoning presented in the beginning of this section, we conclude that; even though is not always the case, yet, as in practice, in many applications (e.g., when information are distributed over a number of symmetric components), if not most, data-flow locations do actually block the systemwide symmetry. Henceforth, when we formally define the notion of component symmetry in the followings, we shall include hiding of (potentially blocking) data-flow locations of symmetric components as part of the requirements.

Considering all aforementioned discussions, here, we define the notion of groups that act on a system as groups that act on the components of the that system such that components considered symmetric by those groups are identical and, in addition, the result of translating that group acting on the components of the system to the level of states of the composite system acts on the result of hiding data-flow locations of symmetric components from constraint automata of the composite system.

Definition 6.5 (Component Symmetry (Groups Acting on a System))**.** Let $S = \{\mathcal{A}_i \mid 1 \leq i \leq n\}$ be a set of n constraint automata where $\mathcal{A}_i = \langle Q_i, \mathcal{N}_i, \mathcal{N}_{i,in}, \mathcal{N}_{i,out}, \rightarrow_i, Q_{0,i}, AP_i, L_i \rangle$. Let $\mathcal{G}$ be a group acting on S for which any two components that are considered symmetric by $\mathcal{G}$ are identical. Then, group $\mathcal{G}$ is said to be acting on S if and only if,

$$\mathcal{G}_{\bowtie} \text{ is acting on } \exists[N_1, \ldots, N_k](\mathcal{A}_1 \bowtie \ldots \bowtie \mathcal{A}_n)$$

where, $\{N_1, \ldots, N_k\} = \bigcup_{i:\ \{\mathcal{A}_i\} \subset \Theta_{\mathcal{G}(\mathcal{A}_i)}} \mathcal{N}_i$, is the set of data-flow locations of all components that have a symmetric counterpart in the system. ■

6.2 Certification and Reduction of Component Symmetry

Given a system, S, together with a group, $\mathcal{G}$, acting on its components, we can certify the provided symmetry without having to build and store the whole system beforehand. To do so, we can simply translate $\mathcal{G}$ to the level of states of the composite system, $\mathcal{G}_{\bowtie}$. To represent $\mathcal{G}_{\bowtie}$, on the other hand, we don't need to compute the whole set of states of the composite system and it would rather suffice to consider the set of states of single member of each orbit of $\mathcal{G}$ and the size of orbits. As an instance, for the system depicted in Figure 6.1, we only need to consider the states of the two nodes and the set of states of an instance of 'alt' component and note that there are 3 instances of 'alt' in the system. This way, we can simply enumerate states of the composite system (or members of each orbit) on the fly – by simply considering k states out of each orbit where k is the size the orbit, e.g., $k = 3$ for 'alt'.

To certify the symmetry on the other hand, we have to consider any pair of orbits of $\mathcal{G}_{\bowtie}$ and make sure that the data constraints between all pair of their members (one from each orbit) are equivalent after hiding data-flow locations of symmetric components. To this end, we consider all pairs $\Theta_{\mathcal{G}_{\bowtie}}(q)$ and $\Theta_{\mathcal{G}_{\bowtie}}(q')$ where q, q' are states of the composite system. Then, we should choose a representative for each of the orbits $rep(\Theta_{\mathcal{G}_{\bowtie}}(q))$ and $rep(\Theta_{\mathcal{G}_{\bowtie}}(q'))$ and for any set of data-flow locations, N, of the composite system, compare (for equivalence) the data constraints between these representatives (after hiding of data-flow locations of symmetric components) for N with the data constraints between any other pair of states (after hiding of data-flow locations of symmetric components) from those orbits (one from each orbit) for the set of data-flow locations N.

For the reduction of component symmetry, the set of states of the result of reduction would simply be the set of orbits of $\mathcal{G}_{\bowtie}$, the set of its initial states would be the set of all those orbits of $\mathcal{G}_{\bowtie}$ that have an initial state of the composite system in them (the tuples where each states is an initial state of some component) and the set of data-flow locations (as well as the set of in and out data-flow locations) of the result of reduction would be that of the composite system except for those that are shared by the symmetric components.

For the transition relation, on the other hand, we have to consider all pairs of orbits of $\mathcal{G}_{\bowtie}$, $\Theta_{\mathcal{G}_{\bowtie}}(q)$ and $\Theta_{\mathcal{G}_{\bowtie}}(q')$ where q, q' are states of the composite system. For each orbit then, we should consider a representative, $rep(\Theta_{\mathcal{G}_{\bowtie}}(q))$ and $rep(\Theta_{\mathcal{G}_{\bowtie}}(q'))$, respectively. Then we should consider a set N of data-flow locations of the composite system $\mathcal{A}_{\bowtie}$. Assuming that $A_1, \ldots, A_k$ are the data-flow locations that belong to symmetric components – those that should be hidden–, the data constraint between $rep(\Theta_{\mathcal{G}_{\bowtie}}(q))$ and $rep(\Theta_{\mathcal{G}_{\bowtie}}(q'))$ for the set of data-flow locations $N' = N \setminus \{A_1, \ldots, A_k\}$ in the result of reduction would simply be:

$$\exists[A_1, \ldots, A_k]\mathfrak{C}^{\mathcal{A}_{\bowtie}}_{rep(\Theta_{\mathcal{G}_{\bowtie}}(q)),rep(\Theta_{\mathcal{G}_{\bowtie}}(q'))}(N)$$

To compute this, on the other hand, assuming $rep(\Theta_{\mathcal{G}_{\bowtie}}(q)) = (q_1, \ldots, q_n)$ and $rep(\Theta_{\mathcal{G}_{\bowtie}}(q')) = (q'_1, \ldots, q'_n)$ where, n is the number of components of the system, we should consider all combinations of components and each combination consider all transitions from q_i to q'_i if the i-th component is in the considered combination. Therefore, we would need to consider (in the worst case) $2^n.tr$ combination of transitions, where tr is the maximum number of transitions for one component.

For the certification of component symmetry, as discussed above, we need to check data constraints between all pairs of states of the composite system which means we have to consider an exponential number of constraints with respect to the number of symmetric components. Later, we will consider a specific layout of components of the system with respect to symmetry group provided for them that allows for the symmetry to be certified by only inspecting a relatively small portion of the system.

On the hand, for the reduction, the number of states in the reduced system is the same as the number of orbits of composite system. Two states of the composite system are considered symmetric by component symmetric if they only differ in the ordering of states of symmetric components. Therefore, we can represent each orbit of $\mathcal{G}_{\bowtie}$ by a tuple of the form $((q_1, n_1), \ldots, (q_l, n_l))$ where l is the number of orbits of $\mathcal{G}$ – the group acting on the components –, q_i belongs to the state space of the components of i-th orbit of $\mathcal{G}$ – all components in each orbit of $\mathcal{G}$ have the same set of states, as they are identical – and n_i is a number between 0 and the number of components in the i-th orbit of $\mathcal{G}$. In plain words, we can label each orbit of $\mathcal{G}_{\bowtie}$ with the number symmetric components in each state (of the respective state space of the symmetric components).

In other words, the number states in the result of reduction, would be proportional to the $(k+1)^n$ where n is the maximum number of states of a component and k is the number of symmetric components. Which means, there will be a polynomial growth in the number of states of the reduced system, instead of the exponential growth in the number of states of the composite system – state space explosion problem –, when the number of symmetric components grow.

Therefore, using symmetry reduction by considering symmetries of components specified in constraint automata can help alleviate the state space explosion problem for constraint automata – as it is widely known and practiced on other formalisms.

As we discussed in Chapter 5, the result of reduction can only be used for checking properties that use those atomic propositions in their temporal logic formulas that the group of symmetry is an invariant group for. In this chapter, we have introduced specific form of

symmetry groups (the result of translation of symmetries) for the case of component symmetry. In particular, two states are symmetric if one is the result of reordering of members of the tuple of the other where only the states of symmetric components can be swapped. Therefore, in the case of component symmetry, we know precisely what atomic propositions the group is invariant for. Namely, those atomic propositions that do not depend on the order of states of symmetric components.

On the other hand, it is usually the case that in the composite system, the atomic propositions are the tuples of atomic propositions of individual components in the same order as their states. In other words, in a system with n components, a state $(q_1, \ldots, q_n)$ is labeled with the set of atomic propositions $L_1(q_1) \times \cdots \times L_n(q_n)$, where L_i is the labeling function of the i-th component.

However, here, we cannot simply label each state with tuples of atomic propositions of the individual states of the tuple. Yet, we can use the same technique that we used for reasoning about the growth of reduced system in the number of symmetric components – counting number of symmetric components in each state, of the symmetric component, in each state of the composite system. In other words, as identical components have the same atomic propositions and labeling function, as well as having the same set of states, we can simply, for each state of the reduced system, count the number of components being labeled to each atomic propositions.

As an example, consider the case where a component has the set of atomic propositions, $\{ap_1, ap_2\}$, and there are n symmetric copies of this component in the system, then we can use atomic propositions $\{(i, ap_1) \mid 0 \leq i \leq n\} \cup \{(i, ap_2) \mid 0 \leq i \leq n\}$ to label the states of the reduced system.

7 Three Tier Component Symmetry

So far, we have defined the notion of symmetry on the level of states of constraint automata (Chapter 5) and lifted it to the case of symmetry over components of a system (Chapter 6). There, we saw that reducing the system with respect to component symmetry helps bring the growth of the states of the system (in the number of symmetric components) from exponential down to polynomial. We discussed that for reduction and certification of component symmetry, we do not need to build and store the whole composite system and we can simply examine or reduce it by considering small portions of the system one after another.

However, in this chapter, we are going to discuss the notion of three tier symmetry, a case of component symmetry where, the components of the system follow a specific layout. In particular, we are going to isolate the part of the system that is directly connected (via data-flow locations) to the symmetric components while we assume that symmetric components have no communications (no shared data-flow locations).

We refer to this isolated part of the system that has direct contact (shared data-flow locations) with the symmetric components as *glue* tier. Glue tier, sits between the tier of symmetric components and the rest of the system. This, as we will discuss in this chapter, allows us to carry out certification of component symmetry with much less effort by only having to examine the glue tier.

On the other hand, this would help make reduction simpler, as we can simply reduce the tier of symmetric components and glue tier together. Since the result of this reduction is bisimilar to the composite system consisting of the symmetry tier and glue tier, we can simply compose the result of reduction with the rest of the system and be sure that the result is bisimilar to the whole composite system.

7.1 Three Tier Layout of Symmetry and Three Tier Component Symmetry

We will use the name three tier component symmetry to refer to the specific case of component symmetry we are going to discuss in this chapter. Given a system and a group acting on the components of that system, they form a three tier component symmetry if they form a three tier layout of symmetry and the provided group acts on the given system. The name three tier layout of symmetry comes from the fact that we divide the system into three tiers, the tier of symmetric components, the tier of the non-symmetric components (which we usually refer to as “the rest of the system”) and the glue tier which mediates communication of

the symmetric components and the rest of the system.

In other words, the components of the symmetric components' tier, do not share any data-flow locations among themselves and with the rest of the system and they only share data-flow locations with glue tier components which could have data-flow locations shared with the rest of the system. On the other hand, two components can have data-flow locations with the same component in the glue tier only if they are symmetric to one another. Figure 7.1 shows the topology of a system in three tier component symmetry.

Definition 7.1 (Three Tier Layout of Symmetry)**.** Let $S = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ be a system and $\mathcal{G}$ be a group acting on its components. Let

$$\mathcal{A}_i = \langle Q_i, \mathcal{N}_i, \mathcal{N}_{i,in}, \mathcal{N}_{i,out}, \rightarrow, Q_{i,0} \rangle$$

and

$$\begin{aligned} Sym &= \{\mathcal{A} \in S \mid \{\mathcal{A}\} \subset \Theta_{\mathcal{G}}(\mathcal{A})\} \\ Glue &= \{\mathcal{A}_i \in S \mid \exists \mathcal{A}_j \in Sym.\, \mathcal{N}_i \cap \mathcal{N}_j \neq \emptyset\} \\ Rest &= S \setminus (Sym \cup Glue) \end{aligned}$$

Then, S and $\mathcal{G}$ form a three tier layout of symmetry (or a three tier layout for short) if and only if we have:

$$\forall \mathcal{A}_i, \mathcal{A}_j \in Sym.\, \mathcal{N}_i \cap \mathcal{N}_j = \emptyset \tag{7.1}$$

$$\forall \mathcal{A}_i, \mathcal{A}_j \in Sym.\ ((\exists \mathcal{A}_k \in Glue.\, \mathcal{N}_i \cap \mathcal{N}_k \neq \emptyset \wedge \mathcal{N}_j \cap \mathcal{N}_k \neq \emptyset) \implies \mathcal{A}_i \in \Theta_{\mathcal{G}}(\mathcal{A}_j)) \tag{7.2}$$

and

$$\begin{aligned} \forall \mathcal{A}_i, \mathcal{A}_j \in Glue.\, \Big((\mathcal{N}_i \cap \mathcal{N}_j \neq \emptyset) \implies \\ (\forall \mathcal{A}_k, \mathcal{A}'_k.\, ((\mathcal{N}_k \cap \mathcal{N}_i \neq \emptyset) \wedge (\mathcal{N}'_k \cap \mathcal{N}_j \neq \emptyset)) \implies \mathcal{A}'_k \in \Theta_{\mathcal{G}}(\mathcal{A}_k)) \Big) \end{aligned} \tag{7.3}$$

■

In the definition of three tier layout of symmetry (Definition 7.1), above, Condition (7.1), specifies that no two components within the symmetry tier can share any data-flow locations. Condition (7.2), specifies that if there are two components that have data-flow locations shared with the same component in the glue tier, then, those two components should be symmetric. And finally, Condition (7.3), specifies that if there are two components in the glue tier that share some data-flow locations among themselves, then any two components in the symmetry tier that share any data-flow locations with either of them should be symmetric. Conditions (7.2) and (7.3) together, specify the fact that the components of the glue tier can be separated into sets in such a way that components of each set only share data-flow locations with components of the rest of the system, other glue components of the same set or a specific orbit of components in the symmetric tier. This fact can be clearly seen in the schematic of Figure 7.1.

In the sequel, we say a system has three tier component symmetry (or three tier symmetry for short) if there is a symmetry group acting on that system such that the system and the group have a three tier layout.

As we already mentioned, the certification and reduction of three tier component symmetries, for many cases if not most, is easier and more efficient. To show this, we will need to introduce two new notions, symmetry support and data-flow symmetry.

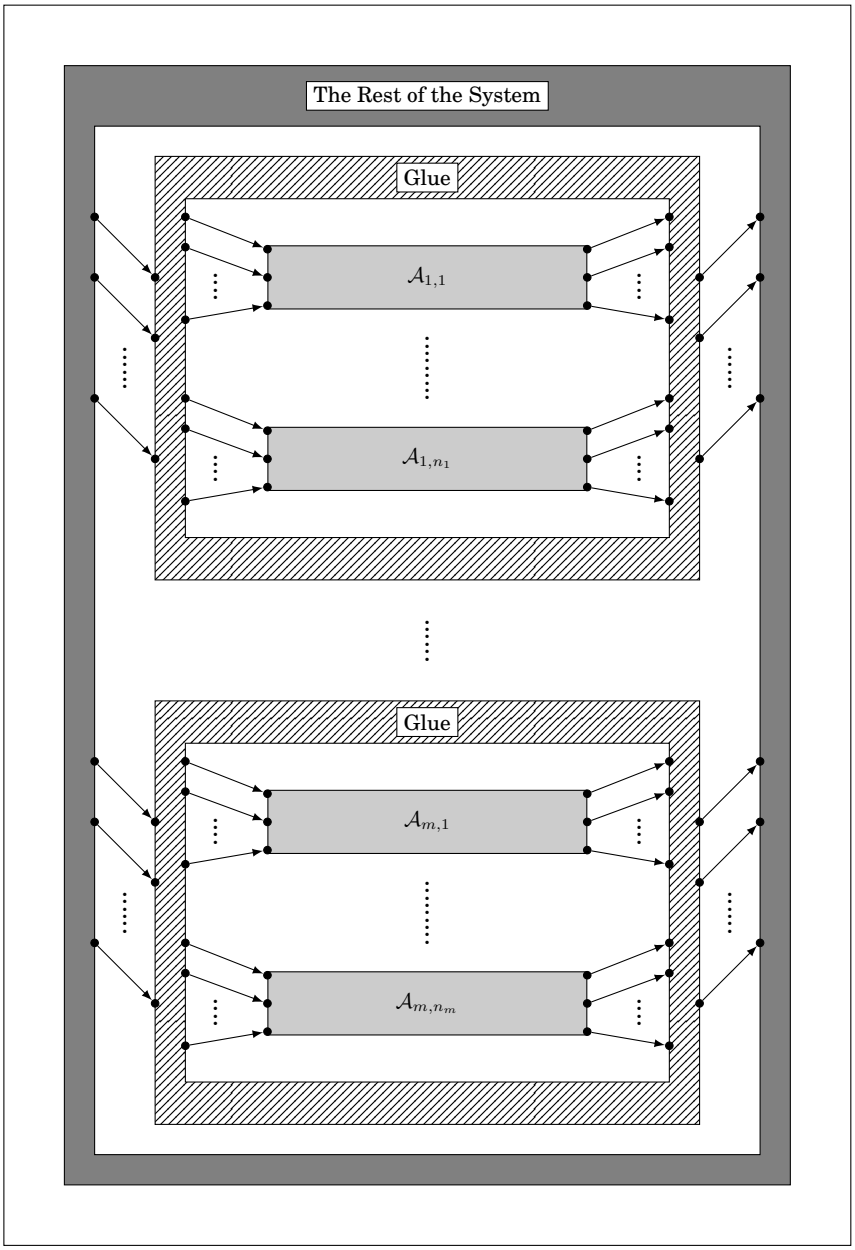


Figure 7.1: The Topography of a System and a Group Acting on that System that Form a Three Tier Layout of Symmetry. Here, for all $1 \leq i \leq m$ and $1 \leq j, k \leq n_i$, we have $\mathcal{A}_{i,j}$ is declared symmetric (by the group) and is identical to $\mathcal{A}_{i,k}$. Data-flow locations are denoted by small black circles on the border of the components (or tiers) they belong to. Two data-flow locations connected by an arrow have the same name while the one on the source of the arrow is in the set of output data-flow location of its constraint automaton and similarly the one at the target of the arrow belongs to the set of input data-flow locations of its respective constraint automaton. Please note that this schematic is not a Reo circuit.

7.2 Symmetry Support and Data-Flow Symmetry

Let's recall the reason why we defined the component symmetry in such a way that hiding of data-flow locations was part of it. It was due to the fact that swapping the states corresponding to symmetric (and hence identical) components in the states of the composite system (which are tuples of states of individual components), would result in having different data-flows (as result of different data constraints) throughout the system which blocked symmetry of the system altogether.

On the other hand, we know that in a system and a group forming a three tier layout, components that are declared symmetric by the symmetry group do not share any data-flow locations with any components that are not in their glue tier. This means that, swapping states of components that are declared symmetric by the symmetry group of the three tier layout, would result in data-flow of the whole system to be different in the values of data-flow locations of symmetric components – and of glue tier as they share data-flow locations with symmetric components.

In this sense, we can say glue layer accepts symmetric data-flow (or it is data-flow symmetric), if, in any state, it accepts a data-flow from symmetric components if and only if it accepts the result of swapping values (of data-flow locations) of that data-flow in any way that corresponds to swapping of symmetric components.

As an instance consider the case where the glue code only consists of Reo nodes (route or replicate). Reo nodes, in any of their states – as they only have one–, accept a value on some source or sink node connected to them if and only if they accept the same value on any other source or sink, respectively. Hence, if there is a component that has a single source and a single sink channel end (e.g., a channel), connecting a number of these components in such a way that all their source channel ends are connected to a single Reo node and all their sinks to another, would result in a system that is symmetric – the translation of component level symmetry that declares all instances of the component at hand symmetric to the level of states of the composite system would act on the composite system after hiding of data-flow locations corresponding to individual source and sink channel ends of the symmetric components.

In order to make use of this phenomenon to make certification of three tier symmetry easier, we define the notions of symmetry support and data-flow symmetry.

7.2.1 Symmetry Support

Consider the case where there are four states q_1, q_2, q_3 and q_4 in a constraint automaton $\mathcal{A}$, such that for some set of data-flow locations of $\mathcal{A}$, N, $\mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N) \not\equiv \mathfrak{C}^{\mathcal{A}}_{q_3,q_4}(N)$. Then a permutation σ, for which we have $\sigma(q_1) = q_3$ and $\sigma(q_2) = q_4$, would not be a symmetry permutation for $\mathcal{A}$.

Although, it can very well be the case that, considering two data-flow locations A_1 and A_2 in $\mathcal{A}$, such that N' is the set obtained from N by renaming A_1 to N_2 and vise versa, would result in having $\mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N)$ equivalent to $(\mathfrak{C}^{\mathcal{A}}_{q_3,q_4}(N))'$, where $(\mathfrak{C}^{\mathcal{A}}_{q_3,q_4}(N))'$ is obtained from $\mathfrak{C}^{\mathcal{A}}_{q_3,q_4}(N)$ by swapping d_{A_1} and d_{A_2}. In such a case, hiding data-flow locations A_1 and A_2 from $\mathcal{A}$ would result in:

$$\mathfrak{C}^{\exists[A_1,A_2]\mathcal{A}}_{q_1,q_2}(N) \equiv \mathfrak{C}^{\exists[A_1,A_2]\mathcal{A}}_{q_3,q_4}(N)$$

Which means, the permutation σ, would be a symmetry permutation for $\exists[A,B]\mathcal{A}$ – provided that other symmetries indicated by σ hold. Here, we generalize this notion, by introducing the notion of symmetry support.

Definition 7.2 (Supported State Symmetry)**.** Let $\mathcal{A}$ be a constraint automaton with the set of states Q and the set of data-flow locations $\mathcal{N}$, $\mathcal{G}$ be a group acting on Q and $\mathcal{H}$ be a group acting on $\mathcal{N}$. Then, $\mathcal{H}$ supports action of $\mathcal{G}$ on constraint automaton $\mathcal{A}$ if and only if we have:

$$\forall \sigma \in \mathcal{G}\ \forall q_1, q_2 \in Q\ \exists \sigma' \in \mathcal{H}\ \forall N \subseteq \mathcal{N}.\ \left(\mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N) \equiv \gamma_{\sigma'}\left(\mathfrak{C}^{\mathcal{A}}_{\sigma(q_1),\sigma(q_2)}(\delta_{\sigma'}(N))\right)\right) \tag{7.4}$$

where, $\gamma_\sigma : \mathfrak{Dcon}(\mathcal{N}) \to \mathfrak{Dcon}(\mathcal{N})$ is the function that maps each d_A in a data constraint to $d_{\sigma(A)}$ and $\delta_\sigma : \mathcal{N} \to \mathcal{N}$ is as follows:

$$\delta_\sigma(N) = \{\sigma(A) \mid A \in N\}$$

■

7.2.1.1 Symmetry Support Leads to Symmetry

As explained earlier, the motivation of introducing the notion of symmetry support is to help us account for the data-flow locations that are blocking the symmetry. Here, we show that if group $\mathcal{H}$ supports the action of a group $\mathcal{G}$ on a constraint automaton $\mathcal{A}$, then, $\mathcal{G}$ is acting on the result of hiding those data-flow locations from $\mathcal{A}$ that according to $\mathcal{H}$ have another data-flow location symmetric to them.

Theorem 7.3 (Supported State Symmetry Leads to Symmetry)**.** Let $\mathcal{A}$ be a constraint automaton with the set of states Q and the set of data-flow locations $\mathcal{N}$. Let $\mathcal{G}$ be a group acting on Q and $\mathcal{H}$ be a group acting on $\mathcal{N}$ such that $\mathcal{H}$ supports action of $\mathcal{G}$ on $\mathcal{A}$. Then, $\mathcal{G}$ is acting on

$$\exists[A_1, \ldots, A_n]\mathcal{A}$$

where,

$$\{A_1, \ldots, A_n\} = \{A \mid \ \{A\} \subset \Theta_{\mathcal{H}}(A)\}$$

is the set of data-flow locations that have symmetric counterpart (apart from themselves) according to $\mathcal{H}$. ■

Proof. It is easy to see that after hiding all those data-flow locations that have some other data-flow locations declared symmetric to them by $\mathcal{H}$, $(A_1, \ldots, A_n)$, we will have:

$$\forall g \in \mathfrak{Dcon}(\mathcal{N})\ \forall \sigma \in \mathcal{H}.\ \gamma_\sigma(g) = g$$

and

$$\forall N \subseteq \mathcal{N}\ \forall \sigma \in \mathcal{H}.\ \delta_\sigma(N) = N$$

It is due to the fact that all remaining data-flow locations (those that are not hidden) are mapped to themselves by any permutation of $\mathcal{H}$.

Therefore, we can rewrite Condition (7.4), from Defition 7.2, as:

$$\forall \sigma \in \mathcal{G}\ \forall q_1, q_2 \in Q\ \forall N \subseteq \mathcal{N}.\ \left(\mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N) \equiv \mathfrak{C}^{\mathcal{A}}_{\sigma(q_1),\sigma(q_2)}(N)\right)$$

Which is the same condition for having group $\mathcal{G}$ acting on $\exists[A_1, \ldots, A_n]\mathcal{A}$. □

7.2.2 Data-Flow Symmetry

Data-flow symmetry is a form of symmetry where the nature of symmetry is witnessed in data-flow locations rather than states of a constraint automaton. In plain words, two data-flow locations are symmetric, if the data-flow that can pass through them in any transition between any two states (the data-flow underlying the constraints between them) is symmetric. More precisely, two data-flow locations A and B are symmetric, if and only if, for any pair of states q and q', for any set of data-flow locations N, the constraints between q and q' for N is equivalent to the swapping d_A and d_B in constraints between q and q for N' where N' is obtained from N by assuming $A \in N'$ if and only if $B \in N$ and $B \in N'$ if and only if $A \in N$. Similarly to the case of symmetry over states, we define the notions of data-flow symmetry permutations, data-flow symmetry groups, etc.

In the sequel, we use DF as a short hand for data-flow, e.g., DF-action instead of data-flow action or DF symmetry instead of data-flow symmetry.

Definition 7.4 (DF Permutations for Constraint Automata)**.** Let $\mathcal{A}$ be a constraint automaton with the set of states Q and the set of data-flow locations $\mathcal{N}$. Then, a DF symmetry permutation $\sigma : \mathcal{N} \to \mathcal{N}$ for constraint automaton $\mathcal{A}$ is a bijective function such that:

$$\forall q_1, q_2 \in Q \; \forall N \subseteq \mathcal{N}.\; \big(\mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(N) \equiv \gamma_\sigma\big(\mathfrak{C}^{\mathcal{A}}_{q_1,q_2}(\delta_\sigma(N))\big)\big)$$

where, γ_σ and δ_σ are as in Definition 7.2. ■

It is easy to see that if σ and σ' are two DF symmetry permutation for a constraint automaton, then, their inverses as well as their compositions are also DF symmetry permutations of that constraint automaton. The proof is very similar to the proof of Theorem 5.2, and is thus omitted.

Definition 7.5 (DF Groups acting on Constraint Automata)**.** Let $\mathcal{A}$ be a constraint automata. Then, a group acting on $\mathcal{A}$ is a group $\mathcal{G} = \langle G, \circ, .^{-1}, id\rangle$ where G is a set of permutations for constraint automaton $\mathcal{A}$.

■

Remark 7.6 (Groups DF Acting on Constraint Automata versus DF Groups Acting on Data-Flow Locations of Constraint Automata)**.** This is very important to note that not any group that is acting on the set of data-flow locations of a constraint automaton is a DF symmetry group of that constraint automaton (DF acting on the automaton). Hence, whenever we say a group is acting on the data-flow locations of a constraint automaton it does not necessarily mean that it is DF acting on the constraint automaton itself. ■

Analogously to the case of symmetry (state symmetry defined in Chapter 5), the problem of checking whether a group that acts on the data-flow locations of a constraint automaton also DF-acts on that constraint automaton is **CoNP**-Complete. The exact problem statement and proof of complexity is very similar to the case of state symmetry (see Definition 5.5 and Theorem 5.6) and are thus omitted. It is yet worth noting that since to confirm DF-action of a group we should certify each of its permutations (or equivalently each permutation of its generating set), the complexity of certifying groups DF-acting on constraint automata is also **CoNP**-Complete.

7.2.3 Data-Flow Symmetry and Symmetry Support under Join

If for a constraint automaton, $\mathcal{A}$, a group $\mathcal{H}$ is supporting the action of another group, $\mathcal{G}$, on $\mathcal{A}$, and there is another constraint automaton $\mathcal{A}'$ such that it has all the data-flow

locations of $\mathcal{A}$ (and possibly more), and there is a group $\mathcal{H}'$ DF-acting on $\mathcal{A}'$ in such a way that permutations of $\mathcal{H}'$ restricted to data-flow locations of $\mathcal{A}$ would result in permutations of $\mathcal{H}$; then, we can obtain a pair of groups $\mathcal{G}''$ and $\mathcal{H}''$ such that $\mathcal{H}''$ supports action of $\mathcal{G}''$ on $\mathcal{A} \bowtie \mathcal{A}'$. Here, $\mathcal{G}''$ would declare a pairs of state of (q_1, q) from $\mathcal{A} \bowtie \mathcal{A}'$ symmetric to another pair of states q_2, q, if and only if, $\mathcal{G}$ declares q_1 and q_2 symmetric.

This means, if we know that there is a group supporting action of another group on a constraint automaton, we can extend this support through joining if we know the supporting group can be extended to a group that DF-acts on the other constraint automaton. Given this result, we can be sure that action support is propagated through joining of components by just examining the DF action on the other constraint automaton that is joining the one that has symmetry support.

Theorem 7.7 (Supporting State Symmetry under Joining)**.** Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two constraint automata such that for $i \in \{1, 2\}$, we have, Q_i, $\mathcal{N}_i$, $\mathcal{N}_{i,in}$ and $\mathcal{N}_{i,out}$ are respectively, the set of states, the set of data-flow locations, the set of input data-flow locations and the set of output data-flow locations of $\mathcal{A}_i$. In addition, $(\mathcal{N}_{1,in} \cup \mathcal{N}_{1,out}) \subseteq \mathcal{N}_2$ and $\mathcal{A}_1$ and $\mathcal{A}_2$ are joinable, i.e.,

$$\mathcal{N}_1 \cap \mathcal{N}_2 = (\mathcal{N}_{1,in} \cap \mathcal{N}_{2,out}) \cup (\mathcal{N}_{1,out} \cap \mathcal{N}_{2,in})$$

In addition, let $\mathcal{G}_1 = \langle G_1, \circ, .^{-1} \rangle$ and $\mathcal{H}_1 = \langle H_1, \circ, .^{-1} \rangle$ be two groups acting on Q_1 and $\mathcal{N}_1$ respectively, such that $\mathcal{H}_1$ supports action of $\mathcal{G}_1$ on $\mathcal{A}_1$ and we have,

$$\forall A \in (\mathcal{N}_{1,in} \cup \mathcal{N}_{1,out}).\ \Theta_{\mathcal{H}_1}(A) \subseteq (\mathcal{N}_{1,in} \cup \mathcal{N}_{1,out})$$

Furthermore, let $\mathcal{H}_2 = \langle \{\zeta_\sigma \mid \sigma \in H_1\}, \circ, .^{-1} \rangle$ be a group DF-acting on $\mathcal{A}_2$ – it is easy to see that $\mathcal{H}_2$ is indeed a group and its proof is thus omitted –, where $\zeta_\sigma : \mathcal{N}_2 \to \mathcal{N}_2$ is obtained from σ as follows:

$$\zeta_\sigma(A) = \begin{cases} \sigma(A) & \text{if } A \in \mathcal{N}_{1,in} \cup \mathcal{N}_{1,out} \\ A & \text{otherwise} \end{cases}$$

Then, there exist groups $\mathcal{H}$ and $\mathcal{G}$, such that $\mathcal{H}$ supports the action of group $\mathcal{G}$ on $\mathcal{A}_1 \bowtie \mathcal{A}_2$. ■

Proof. We take $\mathcal{H}_2 = \langle \{\xi_\sigma \mid \sigma \in \mathcal{H}_1\}, \circ, .^{-1} \rangle$, where $\xi_\sigma : \mathcal{N}_1 \cup \mathcal{N}_2 \to \mathcal{N}_1 \cap \mathcal{N}_2$ is defined as:

$$\xi_\sigma(A) = \begin{cases} \sigma(A) & \text{if } A \in \mathcal{N}_1 \\ A & \text{otherwise} \end{cases}$$

and $\mathcal{G} = \langle \{\Upsilon_\sigma \mid \sigma \in \mathcal{H}_1\}, \circ, .^{-1} \rangle$, where $\Upsilon_\sigma : Q_1 \times Q_2 \to Q_1 \times Q_2$ is defined as:

$$\Upsilon_\sigma(q_1, q_2) = (\sigma(q_1), q_2)$$

Now, we need to show that $\mathcal{H}$ supports the action of $\mathcal{G}$ on $\mathcal{A}_1 \bowtie \mathcal{A}_2$. To do so, we take arbitrary $\Upsilon_\sigma \in \mathcal{G}$ and states $(q_1, q_1'), (q_2, q_2') \in Q_1 \times Q_2$ and show that there exists $\xi_{\sigma'} \in \mathcal{H}$ such that:

$$\forall N \subseteq (\mathcal{N}_1 \cup \mathcal{N}_2).\ \left(\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(q_1,q_1'),(q_2,q_2')}(N) \equiv \gamma_{\xi_{\sigma'}}\left(\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{\Upsilon_\sigma(q_1,q_1'),\Upsilon_\sigma(q_2,q_2')}(\delta_{\xi_{\sigma'}}(N))\right)\right)$$

or equivalently (by expanding application of Υ_σ):

$$\forall N \subseteq (\mathcal{N}_1 \cup \mathcal{N}_2).\ \left(\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(q_1,q_1'),(q_2,q_2')}(N) \equiv \gamma_{\xi_{\sigma'}}\left(\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(\sigma(q_1),q_1'),(\sigma(q_2),q_2')}(\delta_{\xi_{\sigma'}}(N))\right)\right)$$

where, γ_σ and δ_σ are as in Definition 7.2.

To show that $\xi_{\sigma'}$ exists, we take $\xi_{\sigma'} = \xi_{\widehat{\sigma}}$, where, $\widehat{\sigma}$ is the permutation for which we have:

$$\forall N \subseteq \mathcal{N}_1.\ \left(\mathfrak{C}^{\mathcal{A}_1}_{q_1,q_2}(N) \equiv \gamma_{\widehat{\sigma}}\left(\mathfrak{C}^{\mathcal{A}_1}_{\sigma(q_1),\sigma(q_2)}(\delta_{\widehat{\sigma}}(N))\right)\right) \tag{7.5}$$

In other words, $\widehat{\sigma}$ is the permutation supporting permutation σ for constraints between q_1 and q_2 and constraints between $\sigma(q_1)$ and $\sigma(q_2)$. For further clarity, see definition of symmetry support (Definition 7.2). In particular, we have to show that:

$$\forall N \subseteq (\mathcal{N}_1 \cup \mathcal{N}_2).\ \left(\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(q_1,q_1'),(q_2,q_2')}(N) \equiv \gamma_{\xi_{\widehat{\sigma}}}\left(\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(\sigma(q_1),q_1'),(\sigma(q_2),q_2')}(\delta_{\xi_{\widehat{\sigma}}}(N))\right)\right) \tag{7.6}$$

To show this on the hand, we consider that for any $N \subseteq (\mathcal{N}_1 \cup \mathcal{N}_2)$ and any pair of states $q \in Q_1 \times Q_2$ and $q' \in Q_1 \times Q_2$, we have:

$$\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{q,q'}(N) = \left({}_1\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{q,q'}(N) \vee {}_{1,2}\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{q,q'}(N) \vee {}_2\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{q,q'}(N)\right)$$

Where, ${}_1\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{q,q'}(N)$ is the constraints between q and q' as a result of $\mathcal{A}_1$ making a transition alone (Rule (3.2), Definition 3.7), ${}_{1,2}\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{q,q'}(N)$ is the constraints between q and q' as a result of $\mathcal{A}_1$ and $\mathcal{A}_2$ each making a transition simultaneously (Rule (3.1), Definition 3.7) and ${}_2\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{q,q'}(N)$ is the constraints between q and q' as a result of $\mathcal{A}_2$ making a transition alone (Rule (3.3), Definition 3.7). Please note that in case any of these conditions is not possible (e.g., $\mathcal{A}_1$ cannot make a transition alone between q and q'), its corresponding constraint would be $false$.

Hence, to show (7.6), above, we simply consider the following three cases:

Case 1: $\forall N \subseteq (\mathcal{N}_1 \cup \mathcal{N}_2).\ \left({}_1\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(q_1,q_1'),(q_2,q_2')}(N) \equiv \gamma_{\xi_{\widehat{\sigma}}}\left({}_1\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(\sigma(q_1),q_1'),(\sigma(q_2),q_2')}(\delta_{\xi_{\widehat{\sigma}}}(N))\right)\right)$

Case 2: $\forall N \subseteq (\mathcal{N}_1 \cup \mathcal{N}_2).\ \left({}_{1,2}\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(q_1,q_1'),(q_2,q_2')}(N) \equiv \gamma_{\xi_{\widehat{\sigma}}}\left({}_{1,2}\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(\sigma(q_1),q_1'),(\sigma(q_2),q_2')}(\delta_{\xi_{\widehat{\sigma}}}(N))\right)\right)$

Case 3: $\forall N \subseteq (\mathcal{N}_1 \cup \mathcal{N}_2).\ \left({}_2\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(q_1,q_1'),(q_2,q_2')}(N) \equiv \gamma_{\xi_{\widehat{\sigma}}}\left({}_2\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(\sigma(q_1),q_1'),(\sigma(q_2),q_2')}(\delta_{\xi_{\widehat{\sigma}}}(N))\right)\right)$

And use the fact that for any formulas f_1, f_2, f_3, f_1', f_2' and f_3' for which we have $f_1 \equiv f_1'$, $f_2 \equiv f_2'$ and $f_3 \equiv f_3'$, we will have $f_1 \vee f_2 \vee f_3 \equiv f_1' \vee f_2' \vee f_3'$.

Case 1:

Based on definition of join (Definition 3.7), we can see that Case 1, above, assuming $N \subseteq \mathcal{N}_1$, can be rewritten as:

$$\forall N \subseteq \mathcal{N}_1.\ \left(\mathfrak{C}^{\mathcal{A}_1}_{q_1,q_2}(N) \equiv \gamma_{\xi_{\widehat{\sigma}}}\left(\mathfrak{C}^{\mathcal{A}_1}_{\sigma(q_1),\sigma(q_2)}(\delta_{\xi_{\widehat{\sigma}}}(N))\right)\right)$$

Then, since, $\xi_{\widehat{\sigma}}$ and $\widehat{\sigma}$ coincide for $N \in \mathcal{N}_1$ (based on the definition of ξ_σ above), we can rewrite it as:

$$\forall N \subseteq \mathcal{N}_1.\ \left(\mathfrak{C}^{\mathcal{A}_1}_{q_1,q_2}(N) \equiv \gamma_{\widehat{\sigma}}\left(\mathfrak{C}^{\mathcal{A}_1}_{\sigma(q_1),\sigma(q_2)}(\delta_{\widehat{\sigma}}(N))\right)\right)$$

Which holds, according to (7.5), above. In case $N \not\subseteq \mathcal{N}_1$, we will have:

$$\left({}_1\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(q_1,q_1'),(q_2,q_2')}(N) = false\right) \equiv \left(false = \gamma_{\xi_{\widehat{\sigma}}}\left({}_1\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(\sigma(q_1),q_1'),(\sigma(q_2),q_2')}(\delta_{\xi_{\widehat{\sigma}}}(N))\right)\right)$$

which holds trivially.

Case 2:

Based on definition of join (Definition 3.7), we can see that Case 2, above, as:

$$\forall N \subseteq (\mathcal{N}_1 \cup \mathcal{N}_2). \quad \left(\mathfrak{C}^{\mathcal{A}_1}_{q_1,q_2}(N \cap \mathcal{N}_1) \wedge \mathfrak{C}^{\mathcal{A}_2}_{q_1'q_2'}(N \cap \mathcal{N}_2)\right) \equiv \left(\gamma_{\xi_{\widehat{\sigma}}}\left(\mathfrak{C}^{\mathcal{A}_1}_{\sigma(q_1),\sigma(q_2)}(\delta_{\xi_{\widehat{\sigma}}}(N \cap \mathcal{N}_1)) \wedge \mathfrak{C}^{\mathcal{A}_2}_{q_1',q_2'}(\delta_{\xi_{\widehat{\sigma}}}(N \cap \mathcal{N}_2))\right)\right)$$

Which, we can simplify as follows, according to the fact that $\gamma_{\xi_{\widehat{\sigma}}}$ applied to a conjunction, is the same as conjunction applying $\gamma_{\xi_{\widehat{\sigma}}}$ to both conjuncts.

$$\forall N \subseteq (\mathcal{N}_1 \cup \mathcal{N}_2). \quad \left(\mathfrak{C}^{\mathcal{A}_1}_{q_1,q_2}(N \cap \mathcal{N}_1) \wedge \mathfrak{C}^{\mathcal{A}_2}_{q_1'q_2'}(N \cap \mathcal{N}_2)\right) \equiv \left(\gamma_{\xi_{\widehat{\sigma}}}\left(\mathfrak{C}^{\mathcal{A}_1}_{\sigma(q_1),\sigma(q_2)}(\delta_{\xi_{\widehat{\sigma}}}(N \cap \mathcal{N}_1))\right) \wedge \gamma_{\xi_{\widehat{\sigma}}}\left(\mathfrak{C}^{\mathcal{A}_2}_{q_1',q_2'}(\delta_{\xi_{\widehat{\sigma}}}(N \cap \mathcal{N}_2))\right)\right) \tag{7.7}$$

On the other hand, based on the definition of ξ_σ, above, we know that for any $\sigma \in \mathcal{H}_1$, the values of ξ_σ coincides with the values of σ and ζ_σ for data-flow locations in $\mathcal{N}_1$ and data-flow locations in $\mathcal{N}_2$ respectively. Hence, we can rewrite the (7.7), above, as follows:

$$\forall N \subseteq (\mathcal{N}_1 \cup \mathcal{N}_2). \quad \left(\mathfrak{C}^{\mathcal{A}_1}_{q_1,q_2}(N \cap \mathcal{N}_1) \wedge \mathfrak{C}^{\mathcal{A}_2}_{q_1'q_2'}(N \cap \mathcal{N}_2)\right) \equiv \left(\gamma_{\widehat{\sigma}}\left(\mathfrak{C}^{\mathcal{A}_1}_{\sigma(q_1),\sigma(q_2)}(\delta_{\widehat{\sigma}}(N \cap \mathcal{N}_1))\right) \wedge \gamma_{\zeta_{\widehat{\sigma}}}\left(\mathfrak{C}^{\mathcal{A}_2}_{q_1',q_2'}(\delta_{\zeta_{\widehat{\sigma}}}(N \cap \mathcal{N}_2))\right)\right)$$

Which can be easily concluded from (7.8) and (7.9), below:

$$\forall N \subseteq (\mathcal{N}_1 \cup \mathcal{N}_2). \ \left(\left(\mathfrak{C}^{\mathcal{A}_1}_{q_1,q_2}(N \cap \mathcal{N}_1)\right) \equiv \gamma_{\widehat{\sigma}}\left(\mathfrak{C}^{\mathcal{A}_1}_{\sigma(q_1),\sigma(q_2)}(\delta_{\widehat{\sigma}}(N \cap \mathcal{N}_1))\right)\right) \tag{7.8}$$

$$\forall N \subseteq (\mathcal{N}_1 \cup \mathcal{N}_2). \ \left(\left(\mathfrak{C}^{\mathcal{A}_2}_{q_1'q_2'}(N \cap \mathcal{N}_2)\right) \equiv \gamma_{\zeta_{\widehat{\sigma}}}\left(\mathfrak{C}^{\mathcal{A}_2}_{q_1',q_2'}(\delta_{\zeta_{\widehat{\sigma}}}(N \cap \mathcal{N}_2))\right)\right) \tag{7.9}$$

On the other hand, (7.8) easily follows from the fact that $\widehat{\sigma}$ was specifically chosen to be the permutation that supports permutation σ for constraints between q_1 and q_1' in $\mathcal{A}_1$ (see (7.5)). Furthermore, (7.9) easily follows from the fact that $\mathcal{H}_2$ is DF-acting on $\mathcal{A}_2$.

Case 3: For this case, we simply have to consider the fact that according to definition of joining (Definition 3.7), we can rewrite this case as follows, if we consider $N \subseteq \mathcal{N}_2$:

$$\forall N \subseteq \mathcal{N}_2. \ \left(\mathfrak{C}^{\mathcal{A}_2}_{q_1',q_2'}(N) \equiv \gamma_{\xi_{\widehat{\sigma}}}\left(\mathfrak{C}^{\mathcal{A}_2}_{q_1',q_2'}(\delta_{\xi_{\widehat{\sigma}}}(N))\right)\right)$$

Which, according to the fact that values of $\xi_{\widehat{\sigma}}$ and $\zeta_{\widehat{\sigma}}$ coincide for data-flow locations in $\mathcal{N}_2$, can be rewritten as:

$$\forall N \subseteq \mathcal{N}_2. \ \left(\mathfrak{C}^{\mathcal{A}_2}_{q_1',q_2'}(N) \equiv \gamma_{\zeta_{\widehat{\sigma}}}\left(\mathfrak{C}^{\mathcal{A}_2}_{q_1',q_2'}(\delta_{\zeta_{\widehat{\sigma}}}(N))\right)\right)$$

Which is trivial, according to the fact that $\mathcal{H}_2$ DF-acts on $\mathcal{A}_2$. On the other hand, if we assume $N \not\subseteq \mathcal{N}_2$, similarly to Case 1, we will have both sides of equivalence equal to $false$ – as there can be no such transitions – for which the equivalence is trivial for any permutation on data-flow locations.

Putting all three cases together, we get that, (7.6) above holds. Since, we had chosen σ as an arbitrary member of $\mathcal{G}$ and pairs (q_1, q_1') and (q_2, q_2') as arbitrary members of $Q_1 \times Q_2$, we can rewrite, (7.6) as:

$$\forall \sigma \in \mathcal{G} \ \forall (q_1, q_1'), (q_2, q_2') \in Q_1 \times Q_2 \ \exists \sigma' \in \mathcal{H} \ \forall N \subseteq (\mathcal{N}_1 \cup \mathcal{N}_2). \left(\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{(q_1,q_1'),(q_2,q_2')}(N) \equiv \gamma_{\xi_{\sigma'}}\left(\mathfrak{C}^{\mathcal{A}_1 \bowtie \mathcal{A}_2}_{\Upsilon_\sigma(q_1,q_1'),\Upsilon_\sigma(q_2,q_2')}(\delta_{\xi_{\sigma'}}(N))\right)\right)$$

Which is the literal condition for $\mathcal{H}$ supporting action of $\mathcal{G}$ on $\mathcal{A}_1 \bowtie \mathcal{A}_2$ (see Definition 7.2). □

7.2.4 Three Tier Symmetry Certification and Reduction

The notions of symmetry support and data-flow symmetry were introduced to help make certification of component symmetry in case of three tier symmetry easier. To this end, here, we will show that considering symmetry tier in isolation, it is always the case that an instance of symmetry support is evident. Which, according to Theorem 7.7, allows us to certify symmetry by only examining the glue tier. Taking such an approach, we simply consider the group acting on the data-flow locations of the symmetry tier (the one that is supporting a symmetry there) and check whether its extension (the way $\mathcal{H}_2$ was extended from $\mathcal{H}_1$ in Theorem 7.7) is DF-acting on the glue tier.

Theorem 7.8 (The Intrinsic Symmetry Support of Symmetry Tier)**.** Let S be a system and $\mathcal{G}$ be a group acting on its components such that S and $\mathcal{G}$ form a three tier layout. Furthermore, let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be the constraint automata corresponding to the components in symmetry tier, where Q_i and $\mathcal{N}_i$ are, respectively, the set of states and the set of data-flow locations of $\mathcal{A}_i$.In addition, we assume that, for any $\mathcal{A}_i$ and $\mathcal{A}_j$ that are symmetric, $f_{i,j} : \mathcal{N}_i \to \mathcal{N}_j$ is the bijective function that makes $\mathcal{A}_i$ and $\mathcal{A}_j$ identical (see Definition 6.2).

Then, $\mathcal{H}$ supports the action of $\mathcal{G}_{\bowtie_{1,\ldots,n}}$ on $\mathcal{A}_1 \bowtie \ldots \bowtie \mathcal{A}_n$. Where, $\mathcal{G}_{\bowtie_{1,\ldots,n}}$ is the result of translation of $\mathcal{G}$ to the level of states of $\mathcal{A}_1 \bowtie \ldots \bowtie \mathcal{A}_n$ (the tuples are restricted to the states of components in the symmetry tier) and $\mathcal{H} = \left\langle \{\sigma_{i,j} \mid \mathcal{A}_i \in \Theta_{\mathcal{G}}(\mathcal{A}_j)\}, \circ, .^{-1} \right\rangle$ such that $\sigma(i,j) : (\bigcup_{i=1}^{n} \mathcal{N}_i) \to (\bigcup_{i=1}^{n} \mathcal{N}_i)$ is the permutation that maps data-flow locations of $\mathcal{A}_i$ to the data-flow locations of $\mathcal{A}_j$ which is defined as

$$\sigma_{i,j}(A) = \begin{cases} f_{i,j}(A) & \text{if } A \in \mathcal{N}_i \\ A & \text{otherwise} \end{cases}$$

■

Proof. Since S and $\mathcal{G}$ form a three tier symmetry, we know that $\mathcal{A}_1, \ldots, \mathcal{A}_n$ do not share any data-flow locations. Thus, in $\mathcal{A}_1 \bowtie \ldots \bowtie \mathcal{A}_n$, any combination of transitions from any of the components can fire simultaneously (if their respective source and target states are satisfied). In other words, considering the conditions for producing transitions of the product automaton in Definition 3.7, all preconditions are always satisfied if the joining constraint automata don't share any data-flow locations. Therefore, we can easily see the constraints between two states of $\mathcal{A}_1 \bowtie \ldots \bowtie \mathcal{A}_n$, $(q_1, \ldots, q_n)$ and $(q'_1, \ldots, q'_n)$ for any set of data-flow locations $N \subseteq \mathcal{N}$ is as follows:

$$\mathfrak{C}^{\mathcal{A}_1 \bowtie \ldots \bowtie \mathcal{A}_n}_{(q_1,\ldots,q_n),(q'_1,\ldots,q'_n)}(N) = \bigvee_{\emptyset \subset I \subseteq \{1,\ldots,n\}} \bigwedge_{i \in I} \mathfrak{C}^{\mathcal{A}_i}_{q_i,q'_i}(N \cap \mathcal{N}_i) \tag{7.10}$$

In plain words, we can assume any subset of symmetric components, I, are making a transition and for a subset of components making a transition, the constraint of the composite system is simply the conjunction of respective constraints (each with its set of active data-flow locations – $N \cap \mathcal{N}_i$). Please note that if for some $i \in I$ the i-th component does not make any transitions for $N\mathcal{N}_i$, its underlying constraints between q_i and q'_i would be $flase$, which (because of the conjunction) makes the whole constraint between $(q_1, \ldots, q_n)$ and $(q'_1, \ldots, q'_n)$ for I components, $false$ – the set I of components do not make any transitions simultaneously.

To show that $\mathcal{H}$ supports the action of $\mathcal{G}_{\bowtie_{1,\ldots,n}}$ on $\mathcal{A}_1 \bowtie \ldots \bowtie \mathcal{A}_n$, we simply consider an arbitrary pair of states $(q_1, \ldots, q_n)$ and $(q'_1, \ldots, q'_n)$ and an arbitrary permutation σ from $\mathcal{G}_{\bowtie_{1,\ldots,n}}$ and show that there exists a permutation σ' in $\mathcal{H}$ such that:

$$\forall N \subseteq \mathcal{N}.\ \mathfrak{C}^{\mathcal{A}_1 \bowtie \ldots \bowtie \mathcal{A}_n}_{(q_1,\ldots,q_n),(q'_1,\ldots,q'_n)}(N) \equiv \gamma_{\sigma'}\left(\mathfrak{C}^{\mathcal{A}_1 \bowtie \ldots \bowtie \mathcal{A}_n}_{\sigma((q_1,\ldots,q_n)),\sigma((q'_1,\ldots,q'_n))}(\delta_{\sigma'}(N))\right) \tag{7.11}$$

Where, γ_σ and δ_σ are as in Definition 7.2.
Let's assume, without loss of generality, that permutation σ swaps the state of compponets $\mathcal{A}_{i_1}, \ldots, \mathcal{A}_{i_k}$ with states of the components $\mathcal{A}_{j_1}, \ldots, \mathcal{A}_{j_k}$, respectively, for some $k \leq n$ where, for any $1 \leq l \leq k$, we have $\mathcal{A}_{i_l}$ is symmetric $\mathcal{A}_{j_l}$. Then, we show that $\sigma' = \sigma_{i_1,j_1} \circ \cdots \circ \sigma_{i_k,j_k} \circ \sigma_{j_1,i_1} \circ \cdots \circ \sigma_{j_k,i_k}$ makes (7.11), above, hold.

To show this, on the other hand, according to (7.10), above, as γ_σ and δ_σ only alter data-flow locations $\mathcal{N}_{i_l}$ for $1 \leq l \leq k$, we only need to show that:

$$\forall l \in \{1, \ldots, k\}\ \forall N \subseteq \mathcal{N}.\ \left(\mathfrak{C}^{\mathcal{A}_{i_l}}_{q_{i_l},q'_{i_l}}(N \cap \mathcal{N}_{i_l}) \equiv \gamma_{\sigma'}\left(\mathfrak{C}^{\mathcal{A}_{j_l}}_{q_{j_l},q'_{j_l}}(\delta_{\sigma'}(N) \cap \mathcal{N}_{j_l})\right)\right) \tag{7.12}$$

On the other hand, we know that $\mathcal{A}_{i_l}$ and $\mathcal{A}_{j_l}$ are identical and the function mapping data-flow locations of the first to the second and data-flow locations of the second to the firs are f_{i_l,j_l} and f_{j_l,i_l} respectively. In addition, we know that $\delta_{\sigma'}$ maps the data-flow locations of $\mathcal{A}_{i_l}$ to the data-flow locations of $\mathcal{A}_{j_l}$ according to f_{i_l,j_l} and data-flow locations of $\mathcal{A}_{j_l}$ to the data-flow locations of $\mathcal{A}_{i_l}$ according to f_{j_l,i_l}, as we have $\sigma' = \sigma_{i_1,j_1} \circ \cdots \circ \sigma_{i_k,j_k} \circ \sigma_{j_1,i_1} \circ \cdots \circ \sigma_{j_k,i_k}$ (See the definition of $\mathcal{H}$ in theorem statement, above). Similarly, $\gamma_{\sigma'}$ maps the data data-flow locations (d_A for data-flow location A) of $\mathcal{A}_{i_l}$ to the data of data-flow locations of $\mathcal{A}_{j_l}$ according to f_{i_l,j_l} and data of data-flow locations of $\mathcal{A}_{j_l}$ to data of the data-flow locations of $\mathcal{A}_{i_l}$ according to f_{j_l,i_l}. In plain words, $\delta_{\sigma'}$ and $\gamma_{\sigma'}$ simply reverse each others actions, which results in (according to their identity):

$$\forall l \in \{1, \ldots, k\}\ \forall N \subseteq \mathcal{N}.\ \left(\mathfrak{C}^{\mathcal{A}_{i_l}}_{q_{i_l},q'_{i_l}}(N \cap \mathcal{N}_{i_l}) = \gamma_{\sigma'}\left(\mathfrak{C}^{\mathcal{A}_{j_l}}_{q_{j_l},q'_{j_l}}(\delta_{\sigma'}(N) \cap \mathcal{N}_{j_l})\right)\right)$$

Which, trivially, shows that (7.12) holds, which, in turn, shows that (7.11). On the other hand, since we had chosen $(q_1, \ldots, q_n)$, $(q'_1, \ldots, q'_n)$ and σ arbitrarily in (7.11), we can simply conclude the following, which is the literal condition for $\mathcal{H}$ supporting action of $\mathcal{G}_{\bowtie_{1,\ldots,n}}$ on $\mathcal{A}_1 \bowtie \ldots \bowtie \mathcal{A}_n$ (see Definition 7.2):

$$\forall \sigma \in \mathcal{G}_{\bowtie_{1,\ldots,n}}\ \forall (q_1, \ldots, q_n), (q'_1, \ldots, q'_n) \in (Q_1 \times, \ldots, Q_n)\ \forall N \subseteq \mathcal{N}.$$
$$\mathfrak{C}^{\mathcal{A}_1 \bowtie \ldots \bowtie \mathcal{A}_n}_{(q_1,\ldots,q_n),(q'_1,\ldots,q'_n)}(N) \equiv \gamma_{\sigma'}\left(\mathfrak{C}^{\mathcal{A}_1 \bowtie \ldots \bowtie \mathcal{A}_n}_{\sigma((q_1,\ldots,q_n)),\sigma((q'_1,\ldots,q'_n))}(\delta_{\sigma'}(N))\right)$$

Where, Q_i is the set of states of $\mathcal{A}_i$. □

Hence, as a result of Theorem 7.8 and Theorem 7.7, we know that if the group $\mathcal{H}$, as defined in Theorem 7.8, DF-acts on the product of components in the glue tier, the symmetry holds. As a result, to certify three tier symmetry, we only require to check the components in the glue tier.

As we saw, certifying DF-action on a constraint automaton is **CoNP**-Complete. Yet, in spite of this fact, checking glue tier for DF-action is going to be much more efficient than checking the whole system for symmetry, which is also **CoNP**-Copmlete, as we would be analyzing a small fragment of the system. Moreover, as symmetric components in different symmetry orbits do not share any data-flow locations with components of other orbits or

their glue part, the glue part of each of the orbits can be considered separately for certification.

For the reduction of three tier symmetry, the only betterment is in the fact that only parts of the system, i.e., the symmetric tier and glue tier, need to be reduced as well as the fact that each symmetry orbit (of the component symmetry group) can be considered separately, together with its glue part, for reduction. In particular, the fact remains that any number of components that can be making a transition at the same time, which keeps the exponential time required to do the reduction.

On the other hand, in practice, by increasing the number of symmetric components of any orbit, usually – it depends on the design of the system – there is a slight increase in the size of glue tier as it is only mediating data-flow to the rest of the system. Thus, system designer should be careful to make the glue tier as thin as possible to make certification (and also reduction) easier.

8 Application

In this chapter, we are going to discuss how theories that we have developed in this thesis can be used in practice. In particular, we are going to use the idea of three tier symmetry presented in Chapter7. This approach has been implemented and integrated into Vereofy model checking tool (Section 4.2). Reo circuits (Section 4.1), which are used in Vereofy, provide an excellent way to describe three tier symmetry. Glue tier and symmetric components for each orbit (of the group acting on the components of the system) can be represented by a Reo circuit.

In our approach, we are going to use CARML modules to express the symmetric components and define a number of predefined symmetry scenario. Each symmetry scenario, can have its own restrictions on the components that can be used in that scenario and would provide a specific circuitry for the glue tier. In designing a system then, the designer would simply describe the system specifying as a component of the system an instance of some symmetry scenario with some specific component (CARML module) as the symmetric component and the number of its copies in that specific instance of the scenario in question.

As an example, consider the symmetry scenario in which the CARML module that can be used as the symmetric must have a single input and and a single output port. The symmetry scenario then simply distributes the input it receives (on its single route source node) to the input ports of the instance of the symmetric component and gathers their outputs and sends them out (using its single replicate sink node). The system designer can then simply specify that it needs an instance of this scenario having n copies of a channel (e.g., FIFO channel).

Taking this approach, we only need to make sure that for each scenario, the circuitry provided by that scenario (as the glue tier) is in such a way that preserves three tier symmetry – the group $\mathcal{H}$, as defined in Theorem 7.8 DF-acts on it. Furthermore, based on the specific circuitry of the each scenario and the restriction that it puts on the components that can be used as symmetric components, we can use an approach tailored to that scenario to do an efficient reduction.

In the implementation that was done as part of this thesis, we have considered four exemplary and basic, yet practical, scenarios of symmetry. For each of these symmetry scenario, we have provided a suitable reduction method which given an instance of that symmetry scenario, produces a CARML module equivalent to the result of reduction of that instance. In this chapter, we are going to discuss these scenarios and the method developed for their reduction and present some statistics to show effectiveness of taking this approach for Vereofy.

The approach of producing the the result of reduction on the textual level (producing the CARML module of the reduced system in our approach) has been widely practiced and together with the notion of *generic representatives* (the idea of counting the number of sym-

metric components that are in each of the states of one of them to represent the state space of the reduced system), have been shown (see [ET03]) to be more effective in case of symbolic model checking (than the traditional approach of encoding and integrating orbits into BDDS, see [CEFJ96] as an example). This more effectiveness has also been the motivating reason here for taking this approach, as Vereofy internally uses symbolic model checking. For further information on this approach and specifically on the idea of generic representatives and their other applications see [ET03, BDE+12, DMP07, EW03, MDC06, DM06, KNP06].

Before delving into the specifics of these scenarios, we will first discuss flattening of CARML modules (for the components provided as symmetric components) and discuss the general approach to producing the reduced CARML module for these scenarios.

8.1 Flattening CARML modules

As we saw in Section 4.2, in CARML, the state space of the constraint automata is represented with a number of state variables each of which has an associated type which specifies the values that variable can have. For the purpose of symmetry reduction, we are going to use the approach discussed in Section 6.2, i.e., we are going to count the number of symmetric component in each state. Therefore, we need to first come up with an approach to directly enumerate different states, in order to be able to consider, for each state, the number of components in that state.

Therefore, here, we present the approach called *flattening* which flattens the complex structure of variables of a CARML module. During this process, if a CARML module has n states, the states of the CARML module are translated to $\{0, \ldots, n-1\}$. Naturally, to flatten states of a CARML module, its specification of atomic propositions and transitions which are based on conditions stated over state variables. Here we will first describe the process of flattening state variables and atomic propositions and then discuss how transitions can be flattened. In each step of the way, we use parts of the CARML module `fifo` from Subsection 4.2.1 as examples.

The flattening of CARML modules has been implemented in Vereofy on the level of parse tree of the CARML module and supports the whole syntax of CARML (as specified in [BKKa])[1].

8.1.1 Flattening State Variables and Atomic Propositions

To flatten state variables, we follow an inductive approach. We define flattening for integer range and enumeration types and define flattening for structs and tagged unions by assuming that their elements have already been flattened.

Flattening Integer Range For variables of integer range type, `int(a,b)`, we consider the flattened states $0, \ldots, (\mathtt{b} - \mathtt{a})$ where 0 corresponds to `a` and $(\mathtt{b} - \mathtt{a})$ corresponds to b.

Flattening Enumeration For variables of enumeration type, `enum {id_1,...,id_n}`, we consider the flattened states $0, \ldots, (\mathtt{n} - 1)$ where 0 corresponds to `id_1` and $(\mathtt{n} - 1)$ corresponds to `id_n`.

[1] Except for function calls (see [BKKa] for more details) and transition labels (a feature not present in release versions or Vereofy manual yet).

Flattening Struct For variables of struct type,

```
struct {itype_1:  id_1; ... type_n:  id_n;}
```

we consider the flattened states $0, \ldots, ((\prod_{i=1}^{i=\mathtt{n}} c_i) - 1)$, where, i-th element is flattened to $0, \ldots, (c_i - 1)$. In this encoding, the configuration where i-th element is in state $0 \leq s_i \leq c_i$, is computed as:

$$\sum_{i=1}^{\mathtt{n}} s_i \nu_{i-2}$$

where,

$$\nu_i = \prod_{j=1}^{i} c_i$$

And given a state $0 \leq m \leq ((\prod_{i=1}^{i=\mathtt{n}} c_i) - 1)$ of the struct, the state of the $i-th$ element, s_i can be computed as:

$$s_i = div(rem(m, \nu_{i-1}), \nu_{i-2})$$

where, div and rem are integer quotient and remainder functions respectively.

Flattening tagged union For variables of tagged union type,

```
tagged union {itype_1:  id_1; ... type_n:  id_n;}
```

we consider the flattened states $0, \ldots, ((\sum_{i=1}^{i=\mathtt{n}} c_i) - 1)$, where, i-th element is flattened to $0, \ldots, (c_i - 1)$. In this encoding, the configuration where the union has a value of the i-th element which is in state $0 \leq s_i \leq c_i$, is computed as:

$$s_i + \mu_{i-2}$$

where,

$$\mu_i = \sum_{j=1}^{i} c_i$$

And given a state $0 \leq m \leq ((\sum_{i=1}^{i=\mathtt{n}} c_i) - 1)$ of the tagged union, the state of the $i-th$ element, s_i, can be computed as:

$$s_i = m - \mu_{i-2}$$

In this case, if the computed s_i is $c_i \leq s_i$ or $s_i < 0$, we know that m did not have a value of i-th element to begin with.

Flattening State Space For the whole state space of the CARML module, on the other hand, we simply treat it in the same fashion as a struct.

Initial States in Flattened CARML Modules The initial values for state variables are handled similar to the case of conditions specified for atomic propositions in CARML modules and therefore, to represent the set of initial states of the flattened CARML module, we use a set of flattened states.

As an example consider the states of the CARML module `fifo` from Subsection 4.2.1,

```
var:  int(0,1) X;
var:  enum{empty, full} Y := empty;
```

flattened state	X	Y
0	0	empty
1	1	empty
2	0	full
3	1	full

Table 8.1: Flattened states of CARML module `fifo`.

The result of flattening the state space of `fifo` is illustrated in Table 8.1. Here the set of initial flattened states is $\{0, 1\}$.

Flattening Atomic Propositions As flattening of atomic propositions, we consider a set of flattened states to represent them, namely the set of states where the conditions of that atomic proposition holds.

As an example, consider the atomic proposition `EMPTY` defined as follows:

```
ap:  EMPTY <=> Y == empty;
```

For this atomic proposition, we will have the set $\{0, 1\}$ of flattened atomic propositions (see Table 8.1).

8.1.2 Flattening Transitions

For flattening transitions, we should consider their three parts, i.e., *state guards*, *IO guards* and *state assignments*. For flattening of each transition, we should go over all flattened states of the CARML module they belong to. For each flattened state, we should check if the state and IO guards of that transition hold. If so, we should use state assignments to update that state and get another flattened state (which would be the target of the flattened transition). Therefore, since we are considering different flattened states and the fact values of data-flow locations can be used in state assignments, we can have multiple flattened transitions for a single transition of a CARML module.

State Guards For state guards, assuming we are considering firing of that transition from the flattened state st, we can simply extract the state variables mentioned in that state guard from st and use them to evaluate expressions and check comparisons. If a state guard holds for st, we simply go on and check the next one.

IO Guards For IO guards, assuming we are considering firing of the transition to which that IO guard belongs from the flattened state st, and assuming that checking of all state guards for st was successful, we consider two cases, the case where the IO guard only involves constant values, port activations and port values and the case where the IO guard involves state variables of the CARML module. For those IO guards that only involve constant values, port activations and port values, we simply add them to the IO guards of the flattened transitions as they do not depend on the states of CARML module. On the other hand, if an IO guard involves some state variables, before adding it to the IO guards of the flattened transition, the value of those state variables should be replaced with their value extracted from st which will now be considered as a constant value.

State Assignments For state assignments, assuming we are considering firing of the transition for a flattened state st, we consider the set $S_0 = \{st\}$, then for i-th state assignment we compute the set of flattened states S_i that are the result of updating flattened states in S_{i-1} with respect to the i-th state assignment. In other words, assuming the transition has n state assignments, we start with $S_0 = \{st\}$ and after applying first state assignment we get S_1 and repeat this process until we have S_n by applying n-th state assignment to the set of flattened states S_{n-1}. The final result, i.e., the set of flattened states that can be the results of applying state assignments of the transition to st, is the set S_n.

The reason for considering sets of flattened states is due to the fact that evaluating an expression for assignment can result in multiple values. This would happen when we have an expression that involves the value of a port. Since we do not know the value of that port a priori, we should consider all different values that are possible for that port (all members of data type Data). As an example, if we have an assignment like `Y := #A` in the first transition of CARML module `fifo` from Subsection 4.2.1, we should consider two cases (assuming Data is `int(0,1)`), the case where value of `A` is `0` and the case where value of `A` is `1`. For each case on the hand, we should add the IO guard that assures the value of `A` is the one that we have chosen (0 or 1) to the IO guards of the flattened transition.

On the other hand, simple computation of new values according to state assignments is not enough. This is due to the fact that there can be inconsistent state assignments. It can happen by direct or indirect (in case a member of struct or tagged union is assigned a value and subsequently the whole struct or tagged union is assigned a value) overwriting of some value. Vereofy itself ignores overwriting of state variables by state assignments as long as they are consistent. Therefore, after computing flattened transitions, we will consider their state assignments as equalities (handling them as state of IO guards, which can again add some IO guards to the flattened transitions) and make sure that the target states of the flattened states are consistent with all state assignments (which in turn makes sure state assignments are consistent).

As an example of transition flattening consider the transitions of CARML module `fifo` from Subsection 4.2.1.

```
Y == empty -[{A}]-> Y := full & X := #A;
Y == full -[{B} & #B == X]-> Y := empty;
```

For the first transition, state guard '`Y == empty`' is satisfied in flattened state 0. The IO guard '`{A}`', does not involve state variables and is thus simply added to the IO guards of the flattened transitions. For state assignments, if we consider flattened state $S_0 = \{0\}$ and update flattened states with respect to state assignment '`Y := full`', we get the set of flattened states $S_1 = \{2\}$. Afterwards, considering state assignment '`X := #A`', we get the set of flattened states $S_2 = \{2, 3\}$ and we will have to add IO guard '`#A == 0`' for the flattened state $2 \in S_2$ and IO guard '`#A == 1`' for the flattened state $3 \in S_2$. This process results in two flattened transitions:

```
0 -[{A} & #A == 0]-> 2
0 -[{A} & #A == 1]-> 3
```

Yet, we have go through the state assignments once again and assuming there are equalities confirm state assignments once more. This process, in case of the two flattened transitions above, introduces new IO guards and results in:

```
0 -[{A} & #A == 0 & #A == 0]-> 2
0 -[{A} & #A == 1 & #A == 1]-> 3
```

Doing the same process for all transitions of CARML module `fifo`, we will, altogether, get the following transitions:

```
0 -[{A} & #A == 0 & #A == 0]-> 2
0 -[{A} & #A == 1 & #A == 1]-> 3
1 -[{A} & #A == 0 & #A == 0]-> 2
1 -[{A} & #A == 1 & #A == 1]-> 3
2 -[{B} & #B == 0]-> 0
3 -[{B} & #B == 1]-> 1
```

Please note that the second phase of checking state assignments for the second transition does not introduce any auxiliary IO guards, as the single assignment there does not involve port values.

As it is evident in the example, the process of flattening of transitions might introduce repetitive IO guards or might introduce inconsistent IO guards (it did not happen in our example). Therefore, we have added a few levels of syntactic simplification of IO guards as well as satisfiability checking for them to make sure no inconsistent transitions are produced. After going through simplification and satisfiability checking procedures, the flattened transitions of the CARML module `fifo` are as follows:

```
0 -[{A} & #A == 0]-> 2
0 -[{A} & #A == 1]-> 3
1 -[{A} & #A == 0]-> 2
1 -[{A} & #A == 1]-> 3
2 -[{B} & #B == 0]-> 0
3 -[{B} & #B == 1]-> 1
```

8.2 Reduction

As we explained earlier, we are going to use Reo circuits to isolate the glue tier around each set of symmetric components (each orbit of the group acting on the components) and use different scenarios as template for the glue tier. Here, we are going to have a general discussion of reduction of such symmetry scenarios (glue and symmetric components). In particular, we are going to discuss how the state space of the reduced system looks like and how atomic propositions of the reduced system and its transitions are represented in the CARML module of reduced system. We assume that the ports of the reduced system (according to the circuitry of the glue tier of the scenario) has been taken care of.

8.2.1 State Variables and Atomic Propositions of the Result of Reduction

State Variables Here, we assume that the state space of the glue tier has been taken care of and focus on the state space of symmetric components. Assuming we have k copies of a component for which the flattened CARML module has n states, $0 \dots (n-1)$, we will consider n variables `var_0`, ..., `var_`$(\mathtt{n}-1)$ each with the type `int(0,k)`. Basically, we

are counting the number of symmetric components in each of the n states. For each of the n states, there can be between 0 (no component is in that state) to k (all components are in that state). We can easily see that this is an overstatement and there are many states represented this way that are not valid states, e.g., a state where all components are in state i and they are also all in state j for $0 \leq i < j \leq (n-1)$.

We solve this problem by specifying the initial state of the CARML module in such a way that it is a valid state, i.e., some of the components in different states is exactly k, and taking transitions in such a way that when firing in a valid state, always lead to another valid state. Taking this approach results in having all invalid states unreachable and thus neutral to the behavior of the constraint automaton underlying the CARML module of the result of the reduction.

On the other hand, in specifying initial states of the reduced system, we have to consider a restriction on the CARML module being considered as the symmetric component. This is due to the fact that in specification of CARML modules each state variable must either be initialized to a single value or it must be left open (it can have any initial value). If the symmetric component has more that one initial state, however, say states $0 \leq t_0 < t_1 \leq (n-1)$, the initial state of the result of reduction – apart from the specification of initial states of the glue tier–, should be in any way that we have the condition $\texttt{var\_t}_0 + \texttt{var\_t}_1 == \texttt{k}$ satisfied and we have for any i such that we have $i \neq t_0$ and $i \neq t_1$, $\texttt{var\_i} == 0$. Unfortunately, this is not expressible using the current approach for specifying CARML modules – as a state variable either should have a single initial state or be entirely open.

Therefore, in the implementation of symmetry reduction for this thesis, we have put the restriction that CARML modules used as symmetric components should have a single initial state. Yet, the flattening system correctly detects initial states of the provided CARML module and simply generates an error if there are more that one of them. In case facility is provided for specification of initial states as conditions – similar to the approach used for atomic propositions – CARML modules with more than one initial states can be considered as symmetric components with minimal effort.

As an example consider a variation of the CARML module `fifo` where the state variable `X` also has an initial state 0. Assume that we have to reduce a system with k copies of such CARML module as symmetric component. Then, the state variables of the resulting reduced system (only the state variables corresponding to the symmetric components) would be as follows:

```
var: int(0,k) var_0 := k;
var: int(0,k) var_1 := 0;
var: int(0,k) var_2 := 0;
var: int(0,k) var_3 := 0;
```

Atomic Propositions For the case of the atomic propositions of the result of reduction, as we discussed in Section 6.2, we are going to represent them by counting the number of components that have those atomic propositions. Therefore, if there are k copies of the symmetric components, for each atomic proposition `aprop`, we are going to consider $k+1$ atomic properties, to represent $k+1$ cases where there are i symmetric components that have atomic proposition `aprop` (atomic proposition $\texttt{aprop\_cnt\_i}$) for $0 \leq i \leq k$.

Atomic propositions of the flattened CARML modules, similar to the case of initial states, are simply sets of flattened states. Yet, to specify atomic propositions, we can – and should – use conditions over state variables of the CARML module and therefore, we do not have the aforementioned problem that we had with the case of initial states. Hence, assuming

there are k copies of the CARML module `fifo` considered symmetric, the atomic proposition `EMPTY` will be represented by the following atomic propositions in the CARML module of the result of reduction.

```
ap:  EMPTY_cnt_0 <=> var_0 + var_1 == 0;
ap:  EMPTY_cnt_1 <=> var_0 + var_1 == 1;
                     ⋮
ap:  EMPTY_cnt_k <=> var_0 + var_1 == k;
```

8.2.2 Transitions of the Result of Reduction

Transitions of the reduced system, should be constructed in such a way that is compatible with the states of the reduced system. Here, we are going to discuss the way the transitions of the flattened CARML module considered as the symmetric component adapted to state variables of the CARML module of the result of reduction, explained earlier. Similar to the case of state variables of the result of reduction, here, we assume that the transitions of the glue tier are taken care of.

In case a number of (one or more) transitions (of the glue tier or of the flattened CARML module of the symmetric component) have to be firing simultaneously, it is done in the usual way, i.e., their state guards, IO guards and state assignments are simply combined. Please note that in specification of CARML modules activations of ports (data-flow locations) are part of the IO guards and therefore, we do not need to consider that the activation of data-flow locations of transitions firing simultaneously are compatible – if they are not compatible the IO guard of the result will simply be unsatisfiable and hence neglected. Therefore, here, we simply ignore IO guards and focus on dealing with state guards and IO guards of the transitions resulting from simultaneous firing of a number of (one or more) transitions of the flattened CARML module considered as symmetric component.

Here, we assume there are l transitions from a flattened CARML module with flatten states $\{0, \ldots, n\}$ of the form '`st_i_1 -[ ]-> st_i_2`', for $0 \leq i \leq (l-1)$ that should be firing at the same time. To construct the state guards and state assignments for this transition, we compute two values $consumed_j$ and $produced_j$ for each $0 \leq j \leq n$. For j-th flattened state, $consumed_j$ and $produced_j$ are, respectively, the number of times j-th flattened state have been mentioned as the source state and the number of times j-th flattened state have been mentioned as a target flattened state in these l transitions being considered to be firing at the same time.

Based on these two values ($consumed_j$ and $produced_j$) for each flattened states, we can simply compute the state guards and state assignments. For each j for which we have $consumed_j \neq 0$, we add a state guard

$$\text{`}\texttt{var\_j >= } consumed_j\text{'}$$

and for each j for which we have $produced_j - consumed_j \neq 0$, we add the state assignment

$$\text{`}\texttt{var\_j := var\_j + } (produced_j - consumed_j)\text{'}$$

A simple consideration of the way these state guards and state assignments have been chosen, one can simply see, that the sum $\sum_{i=0}^{n} \texttt{var\_i}$ before and after transitions formed as described above is always the same. Hence, if the constraint automaton corresponding to CARML module of the result of reduction is in a valid state before firing of such a transition

is in a valid state, it will also be in a valid (the sum of `var_i` state variables is the same as the number of copies of the symmetric component) state after its firing as well. Hence, we discussed above, we only need to have the initial state be a valid state. This would cause all invalid states be unreachable and therefore, ineffective in the behavior of the constraint automaton corresponding to the CARML module of the result of reduction.

As an example, consider the following three transitions (IO guards are eliminated):

1. `0 -[ ]-> 1;`
2. `1 -[ ]-> 0;`
3. `0 -[ ]-> 0;`

Different combinations of these transitions are as follows:

```
1:      var_0 >= 1 -[ ]-> var_1 := var_1+1;
2:      var_1 >= 1 -[ ]-> var_0 := var_0+1;
3:      var_0 >= 1 -[ ]-> ;
1,2:    var_0 >= 1 & var_1 >= 1 -[ ]-> ;
1,3:    var_0 >= 2 -[ ]-> var_0 := var_0-1 & var_1 := var_1+1;
2,3:    var_0 >= 1 & var_1 >= 1 -[ ]->
         var_0 := var_0+1 & var_1 := var_1-1;
1,2,3:  var_0 >= 2 & var_1 >= 1 -[ ]-> ;
```

8.3 Symmetry Scenarios

Symmetry scenarios, as discussed earlier, are specific conditions on the component that can be considered symmetric and a template circuitry that describes the layout of glue tier for that scenario. Here, we are going to consider four exemplary symmetry scenarios to show how the technique of symmetry reduction developed in this thesis can be effectively used in practice. For each scenario, we describe the restrictions on the components that can be considered symmetric and present the relevant circuitry, discuss the reduction approach for them and finally show some statistics and comparisons between the original system and the result of reduction.

The RSL and CARML code used to provide the circuitry of these scenarios is collected in a library called 'symmetry' library, which can be used to describe symmetries. The code of this library is available in Appendix A. There each scenario is a Reo circuit named `sym_scenario_i<k,T>` where `i` is the number of symmetry scenario, `k` is the number of copies of symmetric component `T`.

In the version of the Vereofy where the symmetry reduction technique has been implemented, specifying the key `--process-symmetries="filename"`, results in the Vereofy looking for Reo circuits of the type `sym_scenario_i<k,T>` in the whole system and to write the result of their reduction as a CARML module with the name `sym_scen_i_red_k_T` to the file specified by `filename`.

As we discussed earlier, Vereofy uses a symbolic approach to model checking. It, in particular, uses BDDs (Binary Decision Diagrams) to represent model and do model checking. The size of BDDs heavily depends on the order binary variables are considered and in some cases there can be an exponential difference between different representations of the same binary function with different ordering of binary variables. On the other hand, finding an ordering that minimizes the size of representation of the function in terms of

BDDs is **CoNP**-Complete ([Bry86]). Therefore, vereofy always takes a fixed predetermined approach to ordering binary variables. Yet, it provides the facility for the user to ask for a *dynamic reordering* which randomly reorders the variables through construction of the BDD representing the constraint automaton to get a smaller BDD. Therefore, here, in the experimental results, we have recorded the BDD size of the model with Vereofy's predefined ordering as well as the size of BDD when it is asked to reorder the variables.

We have experimented with these four scenarios and have presented their results. In order to have a comparison, we have recorded the size of BDD with and without dynamic reordering, number of reachable states in the model and the time it takes for the model, before and after reduction. In addition, we provide the recorded time and number of classes for each bisimulation test (between the result of the reduction and the original system).

8.3.1 Scenario 1

In this symmetry scenario, there are two restrictions on the CARML module that can be considered as symmetric component. First, it should have at least one port (`in` or `out`) and second, it may not have any epsilon transitions, i.e., transitions where there are no ports active.

8.3.1.1 Circuitry

The Reo circuit of this scenario is depicted in Figure 8.1. This figure represents the circuitry of symmetry scenarios 1 and 3, where there are k copies of the symmetric component which has l source nodes and m sink nodes. In this figure, $\mathtt{T}_1 \ldots, \mathtt{T}_k$ are the k copies of symmetric component and $\mathtt{AC}_1 \ldots, \mathtt{AC}_k$ are circuits of type

```
ACTIVITY_CHECKER<c[i].sources + c[i].sinks>
```

The circuit `ACTIVITY_CHECKER<k>` is a circuit with k source nodes and a single sink node and for it, there is an activity (with a non-deterministically chosen value) on the sink node if there is at least one source node active. The `Reader_INF` is a builtin component of Vereofy's builtin library. It has a single source node through which, at each transition reads a value (there is no restriction on the value read).

In this figure, the small solid circles (not to be mistaken with replicate nodes which are bigger) represent channel ends and the lines that connect them to nodes (with arrows that should not be mistaken with the solid triangle of the synchronous channels) mean that those channel ends are parts of these nodes. If the arrow goes from the channel end to the node, that channel end is a sink channel end and if the arrow goes from the node to the channel end, that channel end is a source channel end. In practice, all these channel ends are combined in the respective nodes they are connected to via arrows, yet representing them all with a single node (a solid circle) would render the figure rather difficult to understand.

In the circuitry represented in Figure 8.1, all source and sink nodes of each instance of symmetric component are connected to a unique instance of `ACTIVITY_CHECKER` circuit. Which means, the sink node of the `ACTIVITY_CHECKER` instance corresponding to each instance of the symmetric component will be active if at least one of the source and sink nodes of that instance of symmetric component is active. All sink nodes of all instances of `ACTIVITY_CHECKER` are in turn connected to a single node. This node, in turn is connected to the source node of the `READER_INF`. This means, at each transition of the whole system, there is exactly one instance of the symmetric component making a non-epsilon transition.

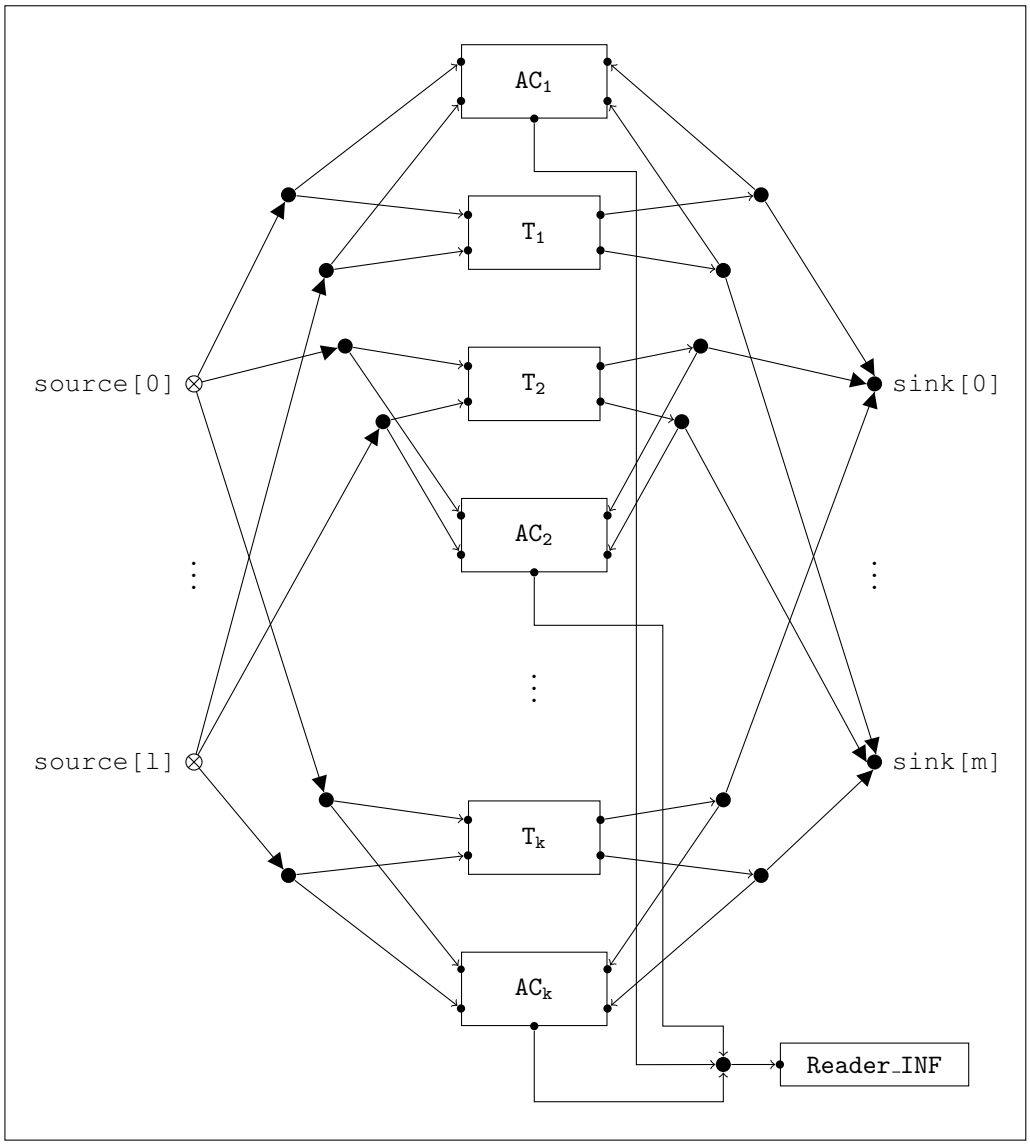


Figure 8.1: The Reo circuit corresponding to symmetry scenarios 1 and 3. For more details see 8.3.1.1.

```
MODULE myfifo1 {
    in:  A;
    out:  B;

    var:  int(0,1) X := 0;
    var:  enum{empty, full} Y := empty;

    Y == empty -[{A}]-> Y := full & X := #A;
    Y == full -[{B} & #B == X]-> Y := empty;
}
```

Figure 8.2: The CARML code used for experimental purposes pertaining to symmetry scenarios 1 and 2.

Copies	BDD nodes	Dynamic Reorder	Reachable States	Build Time
2	78	61	16	0.006s
4	267	189	256	0.006s
10	1536	626	1.04858e+6	0.019s
20	4971	1913	1.09951e+12	0.179s
30	13306	1583	1.15292e+18	0.492s
50	36676	2659	1.26765e+30	2.407s
100	145851	5358	1.60694e+60	20s
150	327526	–	2.03704e+90	418s
500	–	–	–	–
2000	–	–	–	–

Table 8.2: Experimental Results of Symmetry Scenario 1, Before Reduction.

This is due to the structure of the glue tier, as it does not have any transition that does not involve data-flow through symmetric components.

On the other hand, we had the restriction that only those components can be considered as symmetric components of this scenario that have no epsilon transitions. Therefore, in practice, in this scenario, in any transition, there is exactly one instance of the symmetric component making a transition.

Furthermore, a moment's thought on the simple structure of the glue tier of this scenario, one can see that it does indeed satisfy the conditions for three tier symmetry. In other words, in any transition of the glue tier, any value that is accepted on the data-flow locations shared with symmetric components is symmetric with respect to the order of instances of the symmetric component.

8.3.1.2 Reduction

The reduction procedure for this scenario is pretty straightforward, as there is exactly one instance of the symmetric component making a transition at a time. On the other hand, the glue tier has a single state (each of its components have a single state) so we do not need to explicitly represent it. The ports of the CARML module of the reduction can be simply represented with the ports of the symmetric components, as they are each represented by

Copies	Reduction Time	BDD nodes	Dynamic Reorder	Reachable States	Build Time
2	0.002s	530	97	10	0.004s
4	0.001s	2187	180	35	0.005s
10	0.001s	4218	448	286	0.007s
20	0.001s	4093	479	1771	0.006s
30	0.001s	5202	411	5456	0.005s
50	0.001s	6186	617	23426	0.005s
100	0.001s	6061	3679	17851	0.006s
150	0.002s	8154	1079	585276	0.006s
500	0.002s	8029	1887	2.10843e+7	0.007s
2000	0.007s	9445	10386	1.33734e+9	0.010s

Table 8.3: Experimental Results of Symmetry Scenario 1, After Reduction.

Copies	Time	Number of Classes
2	0.005s	6
4	0.02s	15
10	0.776s	66
20	40s	231
30	783s	496
50	–	–
150	–	–
500	–	–
2000	–	–

Table 8.4: Experimental Results of Symmetry Scenario 1, Bisimulation.

a source or sink node in the circuit of the scenario. Therefore, in the CARML module of the result of reduction, we simply need to put ports of the symmetric components and consider its flattened states, its atomic propositions and for transitions of the CARML module of the result of reduction, consider cases where there is exactly one of its transitions being fired.

For experimentation with this symmetry scenario, we have used different number of copies of the CARML module presented in Figure 8.2. Tables 8.2, Tables 8.3 and Tables 8.4, illustrate statistics pertaining to before, after and bisimulation statistics for different copies, respectively. For further discussions on statistics presented for this scenario, as well as other scenarios, refer to Section 8.4.

8.3.2 Scenario 2

Symmetry scenario 2, restricts the CARML modules that can be used with this scenario as the symmetric component the same way as scenario 1. Namely, such a CARML module should have at least one port (`in` or `out`) and it should not have any epsilon transitions, i.e., transitions where there are no ports active.

8.3.2.1 Circuitry

The circuitry of this scenario, depicted in Figure 8.3, is very similar to that of scenario one, explained in 8.3.1.1. In this circuitry, instead of connecting all source and sink nodes of each symmetric component to an instance of `ACTIVITY_CHECKER`, there are two instance of `ACTIVITY_CHECKER` for each instance of symmetric component. For each instance of symmetric component, its source nodes are connected to a unique instance of `ACTIVITY_CHECKER` while its sinks are connected to another unique instance of `ACTIVITY_CHECKER`. All instances of `ACTIVITY_CHECKER` that gather source nodes of symmetric components have their single sink node connected to a unique node which in turn is connected to the source node of an instance of `READER_INF`. All instances of `ACTIVITY_CHECKER` that gather sink nodes of symmetric components have their single sink node treated similarly, i.e., they are connected to a unique node that is connected to a `READER_INF`.

Therefore, in practice, there can be at most one instance of symmetric component making a transition that involves its source nodes and at most one instance that making a transition that involves its sink nodes – it can be the case that these two are the same instance making a transition involving both its source nodes and sink nodes.

On the other hand, we have that the CARML module considered as symmetric component must have no epsilon transitions. As a result, in practice, in any transition of the system, there can be one instance of symmetric component making a transition or there can be two of them, one making a transition involving its source nodes and the other making a transition involving its sink channel ends.

In addition, we can easily see, that the glue tier considered for this scenario satisfies the conditions for a glue that supports three tier symmetry.

8.3.2.2 Reduction

For the reduction technique for this symmetry scenario, similarly to scenario one, we have to consider ports of the CARML module considered as symmetric component as the ports of the CARML module of the result of reduction. Furthermore, we have to consider states and atomic propositions as explained in Sub-section 8.2.1. For the case of transitions, we have to consider all transitions of the CARML module considered as symmetric component, as in scenario 1.

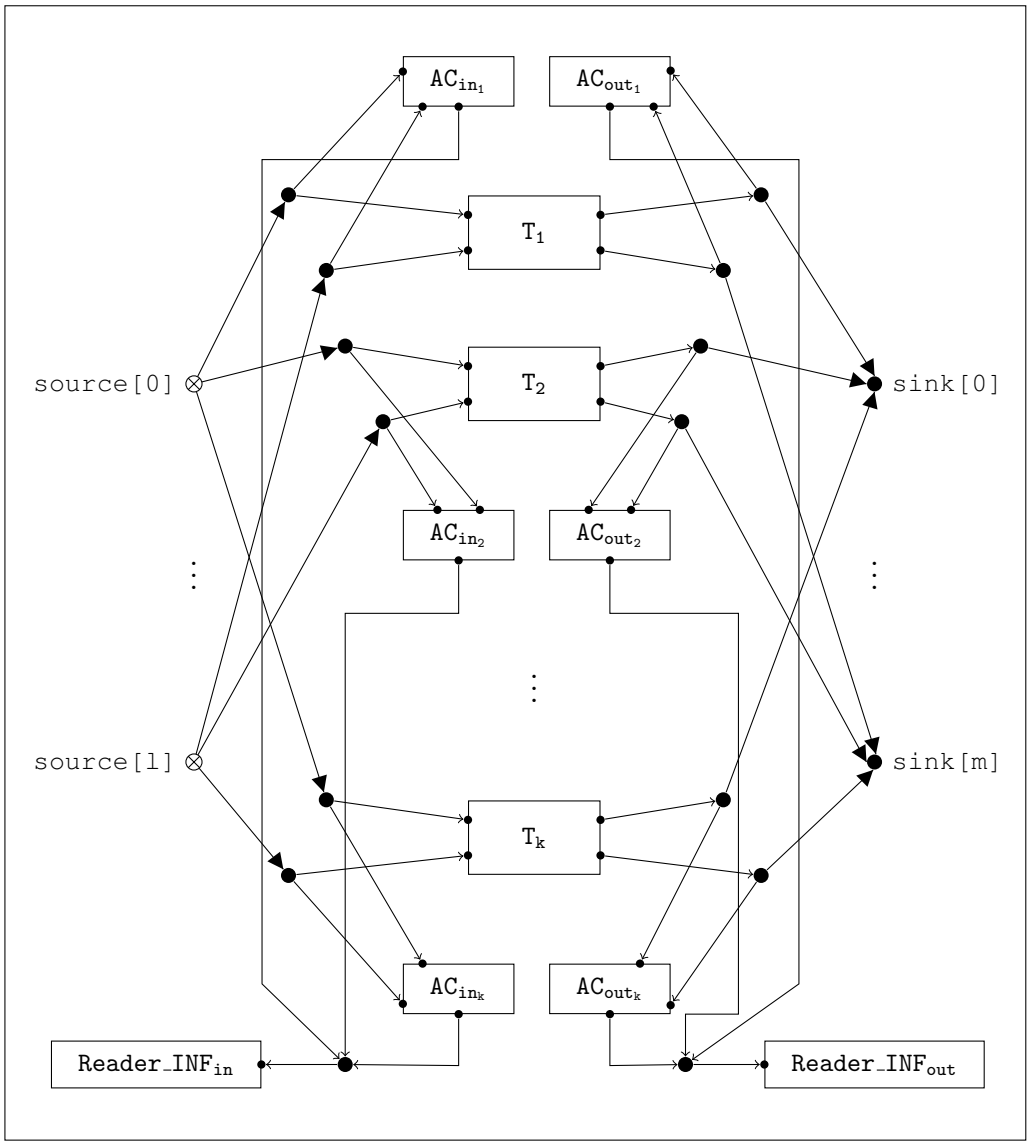


Figure 8.3: The Reo circuit corresponding to symmetry scenarios 2 and 4. For more details see 8.3.2.1.

Moreover, we have to consider cases where there are two of them running in parallel. Yet, we have to take care of the IO guards of such transitions. To do so, we will rename the names of ports in IO guards of the two transitions being considered as running in parallel. Then, assume that in one of them the ports corresponding to renaming of `in` ports are inactive (by adding IO guards specifying such inactivities) and assume that the ports corresponding to the result of renaming of `out` ports in the other transition are inactive. Then, make sure that such a transition as the result of running these two transitions in parallel would have a satisfiable IO guard (by conjoining IO guards and checking its satisfiability). In case the result is satisfiable (these two transitions can indeed run in parallel), the ports that where assumed to be inactive in both transitions are hidden from the conjunction (the same process as used in hiding data-flow locations of the constraint automata). The resulting IO guard, then, after having its ports renamed back to the original names is considered as the IO guard of having those two transitions run in parallel.

As an example, consider the first and second transitions of CARML module `myfifo1` depicted in Figure 8.2. We will first rename the IO guards of these transitions accordingly, which would result in them to be, respectively, as follows:

```
{A1}
{B2} & #B2 == X
```

In this renaming, in the IO guard of the first transition, we have `A` is renamed to `A1` and `B` is renamed to `B1` and in the IO guard of the second transition, we have `A` is renamed to `A2` and `B` is renamed to `B2`. In this case, before continuing, we have to expand the port activity constraints, which would result in:

```
A1 & !B1
B2 & #B2 == X & B1
```

Next, we will make `B1` in the first one and `A2` in the second one in active. Which results in:

```
A1 & !B1 & !B1
B2 & #B2 == X & !A2 & !A2
```

Then, we check if their conjunction,

```
A1 & !B1 & !B1 & B2 & #B2 == X & !A2 & !A2
```

is satisfiable. In this case it is, but, had we chosen to make `A1` inactive in the first one and `B2` in the second one, their conjunction (and even individually), unsatisfiable. Afterwards, we hide `B1` and `A2` from the conjunction which results in:

```
A1 & B2 & #B2 == X
```

Afterwards, we rename the ports back and we will have:

```
A & B & #B == X
```

For experimentation with this symmetry scenario, we have used different number of copies of the CARML module presented in Figure 8.2. Tables 8.5, Tables 8.6 and Tables 8.7, illustrate statistics pertaining to before, after and bisimulation statistics for different copies, respectively. For further discussions on statistics presented for this scenario, as well as other scenarios, refer to Section 8.4.

Copies	BDD nodes	Dynamic Reorder	Reachable States	Build Time
2	168	111	16	0.005s
4	855	445	256	0.008s
10	10083	1155	1.04858e+6	0.024s
20	75223	2413	1.09951e+12	0.336s
30	249363	2740	1.15292e+18	1.569s
50	1140643	4674	1.26765e+30	16s
100	–	–	–	–
150	–	–	–	–
500	–	–	–	–
2000	–	–	–	–

Table 8.5: Experimental Results of Symmetry Scenario 2, Before Reduction.

Copies	Reduction Time	BDD nodes	Dynamic Reorder	Reachable States	Build Time
2	0.002s	950	144	10	0.006s
4	0.002s	4185	271	35	0.008s
10	0.002s	10030	434	286	0.007s
20	0.002s	8250	593	1771	0.010s
30	0.002s	12186	663	5456	0.010s
50	0.002s	14342	798	23426	0.009s
100	0.003s	12562	951	176851	0.013s
150	0.009s	18654	1866	585276	0.09s
500	0.003s	16874	1399	2.10843e+07	0.010s
2000	0.002s	19602	1658	1.33734e+09	0.014s

Table 8.6: Experimental Results of Symmetry Scenario 2, After Reduction.

Copies	Time	Number of Classes
2	0.003s	6
4	0.03s	15
10	1.727s	66
20	153s	231
30	4134s	496
50	–	–
100	–	–
150	–	–
500	–	–
2000	–	–

Table 8.7: Experimental Results of Symmetry Scenario 2, Bisimulation.

```
MODULE myfifo2 {
    in:  A;
    out:  B;

    var:  int(0,1) X := 0;
    var:  enum{empty, full} Y := empty;

    Y == empty -[{A}]-> Y := full & X := #A;
    Y == full -[{B} & #B == X]-> Y := empty;
    Y == full -[ epsilon ]-> Y := empty;
}
```

Figure 8.4: The CARML code used for experimental purposes pertaining to symmetry scenarios 3 and 4.

Copies	BDD nodes	Dynamic Reorder	Reachable States	Build Time
2	85	75	16	0.005s
4	272	217	256	0.005s
10	1433	589	1.04858e+06	0.028s
20	5368	1449	1.09951e+12	0.028s
30	11803	1556	1.15292e+18	0.58s

Table 8.8: Experimental Results of Symmetry Scenario 3, Before Reduction.

8.3.3 Scenario 3

This scenario, is very similar to the first scenario, Section 8.3.1. The scenarios circuitry is exactly the same as scenario 1 and the only difference is in the fact that the CARML module of the symmetric component may have epsilon transitions.

For reduction technique used for this scenario, we simply follow the technique used for scenario 1. Afterwards, for each transition, we assume that there are at most $k-1$ other epsilon transitions running in parallel with it, where k is the number of copies of the symmetric component. As a result, we will have an exponential blow up in the size of the CARML module of the result of reduction.

For experimentation with this symmetry scenario, we have used different number of copies of the CARML module presented in Figure 8.4. Tables 8.8, Tables 8.9 and Tables 8.10, illustrate statistics pertaining to before, after and bisimulation statistics for different copies, respectively. For further discussions on statistics presented for this scenario, as well as other scenarios, refer to Section 8.4.

8.3.4 Scenario 4

This scenario, is very similar to the second scenario, Section 8.3.2. The scenarios circuitry is exactly the same as scenario 2 and the only difference is in the fact that the CARML module of the symmetric component may have epsilon transitions.

For reduction technique used for this scenario, we simply follow the technique used for scenario 2. Afterwards, for each transition produced there, we assume that there are at

Copies	Red Time	File Size	BDD nodes	Dyn Reorder	Reach St	Build Time
2	0.001s	2KB	1024	156	10	0.008s
4	0.002s	7KB	7492	738	35	0.008s
10	0.005s	46KB	27820	2041	35	3.310s
20	0.022s	187KB	47383	12727	1771	1.139s
30	0.052s	426KB	46899	16904	5456	3.489s

Table 8.9: Experimental Results of Symmetry Scenario 3, After Reduction.

Copies	Time	Number of Classes
2	0.008s	6
4	0.054s	15
10	2.767s	66
20	250s	231
30	–	–

Table 8.10: Experimental Results of Symmetry Scenario 3, Bisimulation.

most $k-1$ (in case the transition in question is from a single instance of symmetric components) or $k-2$ (in case the transition in question is from the result of two transitions each from an instance of symmetric components running in parallel) other epsilon transitions running in parallel with it, where k is the number of copies of the symmetric component. As a result, we will have an exponential blow up in the size of the CARML module of the result of reduction.

For experimentation with this symmetry scenario, we have used different number of copies of the CARML module presented in Figure 8.4. Tables 8.11, Tables 8.12 and Tables 8.13, illustrate statistics pertaining to before, after and bisimulation statistics for different copies, respectively. For further discussions on statistics presented for this scenario, as well as other scenarios, refer to Section 8.4.

8.4 Discussion

Before going into details of comparison and contrast of the experimental results, we should mention that all these results were obtained on author's personal computer, which is a MacBook Pro with 2.8 GHz intel Core i7 CPU and 4 GB of DDR3 ram running Mac OS X

Copies	BDD nodes	Dynamic Reorder	Reachable States	Build Time
2	183	114	16	0.005s
4	833	368	256	0.006s
10	9157	1003	1.04858e+06	0.006s
20	66337	2584	1.09951e+12	0.835s

Table 8.11: Experimental Results of Symmetry Scenario 4, Before Reduction.

Copies	Red Time	File Size	BDD nodes	Dyn Reorder	Reach St	Build Time
2	0.001s	4KB	1350	170	10	0.008s
4	0.004s	21KB	10562	755	35	0.091s
10	0.031s	176KB	38898	2454	286	0.091s
20	0.090s	783KB	–	–	–	–

Table 8.12: Experimental Results of Symmetry Scenario 4, After Reduction.

Copies	Time	Number of Classes
2	0.004s	6
4	0.062s	15
10	5.667s	66
20	–	–

Table 8.13: Experimental Results of Symmetry Scenario 4, Bisimulation.

10.8.3. Furthermore, the results were obtained using the *release*[2] mode compilation of the experimental version of Vereofy model checking tool.

The columns that are missing in experimental results are due to the fact that we were unable to perform them as they took longer than they could be waited for. Computations are interrupted after they took more than about half an hour to an hour and had not yielded any results. Furthermore, please note that times specified here are build times of the system and other operations, e.g., computing the number of reachable states, usually took longer than building.

In addition, in case of scenarios 3 and 4, since, as it is noticeable in tables, produced files containing result of reduction grew exponentially, we were not able to go beyond 20 or 30 copies as Vereofy's parser would take considerably long time to parse the files. This is an intrinsic shortcoming of the parser module of Vereofy which takes exponentially long time to parse a file proportional to its size. There is size limit at about 300 to 400 KB, where parsing itself takes more between half an hour to an hour and after that it is much longer – we have not been able to get it to parse files of size 600 KB or larger.

Results of First and Second Scenarios The results of these two scenarios follow almost the same trends. In all cases, the number of reachable states, as expected, is considerably smaller than that of the original states. This is so much that for the case with 2000 copies, we have a difference of the order of 10^{21}. Similarly, in cases large enough, more than 50 copies, we see a huge difference between the time it takes for the original system's BDD to be built and the time it takes for the reduced system. It is so much that for the cases with more than 150 copies in case of the first scenario (even in this case reordering took too long) and cases with more than 50 copies in case of the second scenario was not feasible, as they took too long. Yet, for the reduced version, even with 2000 copies we have 0.010 seconds and 0.014 seconds for first and second scenarios, respectively.

For the case of the size of BDDs on the other hand, we see that for the cases with very few copies, two copies and four copies here, the result of reduction, in fact, has a worse size.

[2]Release mode compilations of program codes, are usually noticeably faster (about twice as fast in this case) than debug mode compilations.

But, moving to cases with more copies, we see that the result of reduction, specially after reordering does a much better job compared to the original system.

Results of Third and Fourth Scenarios As in these scenarios, we have allowed existence of epsilon transitions for the symmetric components, we have that the number of possible transitions in the system and thus the size of file where the result of reduction is written to grows exponentially in the number of copies. This, unfortunately accompanied with inefficiency of Vereofy's parser, prevents us from experimenting with large numbers of copies.

Although, one can notice that the growth in the size of BDDs and build times in these two scenarios are not as fast as they are for the original system. Thus, it is expected that the statistics for the result of reduction would take over the ones for original systems and be better for larger copies of the symmetric component. Yet, unfortunately, this can not be experimentally observed at the moment.

Overall Comparison It should be noted that these scenarios correspond to two, almost extreme cases. On the one hand, scenarios one and two only consider cases where only one or two of the symmetric components are making simultaneous transitions, and on the other hand, scenario three and four allow for any number of symmetric components make transitions (due to epsilon transitions) independently of one another. Therefore, depending on the circuitry and restrictions a symmetry scenario puts, one should expect a result in between these two extremes, in most cases.

9 Conclusion

To conclude this thesis, in this chapter, we present an overview of contributions and results of this thesis, discuss how this line of research may be continued and finally discuss a few points that can help make Vereofy better suited for symmetry reduction.

9.1 Contributions and Results

We started by defining the well known notion of symmetry for constraint automata and showed that the result of reduction with respect to such symmetries is bisimilar to the original system, if only those atomic propositions are considered that are preserved under symmetry. Furthermore, we showed that certifying an instance of symmetry for constraint automata is **CoNP**-Complete.

Secondly, we took advantage of the fact that constraint automata are in practice used to realize component based systems and hence lifted the notion of symmetry to the level of components. There, we restricted symmetric components such that two components can be symmetric if they only differ in the name of their data-flow locations. This way, symmetry could easily be witnessed in the high-level design of the system and be provided by the system designer. In addition, we discussed that certification and reduction of symmetry specified on the level of components of the system can be carried out without having to build the whole system. Yet, the **CoNP**-Completeness of the certification problem persists, as the certification of component symmetry was done through examining the states of the composite system and its transitions.

To make certification and reduction of symmetry easier, in a further step, we divided the components in the system into three tiers. Tier of symmetric components, tier of non-symmetric components and a tier in between called glue which mediates communications between these two tiers and specified a number of restrictions on data-flow locations shared among components both within individual tiers and between them. Moreover, we discussed that this isolation of the components that provide mediation of data-flow between symmetric components and non-symmetric components, allowed for much more efficient certification of symmetry. This was done through checking a form of symmetry over the data-flow locations of the glue tier components that are shared with symmetric components. Even though, certifying the symmetry of data-flow locations is **CoNP**-Complete as well, this time it is only done for a small fragment of the system. On the other hand, it also helped in reduction, as it would allow for considering only reduction of the symmetric components and the glue tier components.

Later, we discussed how these notions can be used in practice. To this end, we considered that for a number of components that are symmetric, they are contained in a Reo

circuit with their glue tier. There, we introduced symmetry scenarios. Each symmetry scenario described a specific circuitry to be used as glue tier and put specific restrictions on the components that can be considered as symmetric components. In practice, the circuitry of the glue tier for each scenario can be certified as preserving three tier symmetry just once. Hence, we could be sure that putting any component in the circuitry of that scenario would result in a symmetric system, as long as that component is compliant with the restriction that that scenario puts on the components which can be considered as symmetric components for that scenario. On the other hand given the properties of the circuitry and restrictions of a symmetry scenario, we could have a tailor-made technique for reduction of symmetries following that scenario.

Finally, we presented four exemplary symmetry scenarios and discussed how one can go around reducing symmetries expressed using each scenario. Later, we presented some experimental results of using those symmetry scenarios and discussed their results.

There, we saw how restrictions on the symmetric components and glue tier circuitry can affect effectiveness of symmetry reduction. We saw that symmetry reduction can be very effective in the case that the number of components that can make simultaneous transitions is limited. On the other hand, we saw that allowing for symmetric components to make epsilon transitions which in turn would result in arbitrary number of components to be able to make a transition in parallel, makes the result of symmetry reduction be even worse than the original system. This was evident in the size of BDDs and build time of the model compared to that of the original system, at least for smaller number of copies, as we were unable to experiment with large number of copies due to exponential growth in the size of the result of reduction and inefficiency of Vereofy's parser module. Although, even those cases, we can see that the growth in BDD size of the model and build time are slower compared to that of the original system, for those small number of copies that we were able to test for. Hence, we expect, but could not experimentally confirm, that for larger numbers of copies, the result would be better for the reduced system compared to the original one.

9.2 Further Research

There can be many ways to continue the line of research for applying symmetry reduction to constraint automata. The most straightforward one being looking for more useful symmetry scenarios and discuss their applicability. On the other hand, the condition we put on the glue tier in three tier symmetric is that its data-flow locations that are shared with symmetric components should be perfectly symmetrical with respect to swapping of data-flow locations shared with each symmetric component. This, of course, is a very tight restriction. As an example, consider the case where a glue tier distributes inputs over symmetric components based on the values of input. Such a system would still show symmetry. A closer inspection of such glue tiers can also be interesting.

On the other hand, one can see if the approach presented in [SG04] can be adopted and used in the context of symmetry for constraint automata. The approach, presented in [SG04], annotates the transitions of the reduced system with information on how the reduced system can be unwound to get the original system. This way, we would be able to use symmetry reduced system to check properties that are not preserved under symmetry.

Furthermore, it would be of interest to see if the approach presented in [DM05] can be adapted to the context of symmetry reduction for constraint automata. In [DM05], authors discuss how using a channel based model of communication we can automatically detect symmetries in a system. This is of importance, particularly for constraint automata as we

have Reo circuits and channels which provide a channel based communication framework which can be taken advantage of for such automatic analysis.

Finally, as advised by [Pel09], it would be of interest to see how symmetry reduction can be combined with other methods of attacking state space explosion problem. Here, we have integrated symmetry reduction into Vereofy, which uses symbolic model checking. Another useful combination can be adding slicing techniques to it. In slicing, only parts of the model that are relevant are considered for model checking. In particular, it is of interest when dealing with models that are obtained from program codes. Yet, here, we have CARML language which is a variation of guarded command language. The structure of CARML modules, and in particular the way state space is described there can benefit a lot from such an approach. On the other hand, there can be systems that are in general symmetric but once some parts of them is eliminated, e.g., as a result of slicing, symmetry can appear which could be made use of for further reduction of the system.

9.3 Vereofy and Symmetry Reduction

Here, we are going to discuss how Vereofy and the symmetry reduction approach implemented in it can be improved for better symmetry reduction.

Most important of all, there is no way in CAEML modules to specify multiple initial states other than leaving the value of a variable free. Therefore, as we discussed in 8.2.1, we are not able to use symmetry reduction for the case of symmetric components with more than one initial state. Should there be a way to express initial states with conditions on the state variables, e.g., similar to the way atomic propositions are expressed in CARML, we can use it to facilitate reduction of components with more than one initial state.

Moreover, as we discussed in the reduction of second symmetry scenario, 8.3.2.2, we are doing a syntactic manipulation of IO guards to get the IO guards of a transition that is the combination of two transitions running in parallel. To this end, to achieve further simplification, and for a more efficient satisfiability checking for IO guards, we can use BDDs which already exist in Vereofy. We did not get involved in this in our brief experimental implementation.

Furthermore, if there was some facility for specifying which transitions can run in parallel, we could simply consider transitions where there are only one symmetric component making a transition and specify those comprising of more than one symmetric components making simultaneous transitions as combinations of them. Should there be such a mechanism, we would not have the exponential blow up in the size of CARML modules representing results of symmetry reduction. In addition, there might even be more efficient ways to construct the BDD of the reduced system; perhaps alleviating the huge BDD sizes for the reduced system of instances of symmetry scenarios three and four.

Bibliography

[AB09] S Arora and B Barak. *Computational Complexity: A Modern Approach*. Cambridge Univ. Press, 2009.

[Ake78] S B Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 100(6):509–516, 1978.

[Arb04] F Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(03):329–366, 2004.

[BB08] T Blechmann and C Baier. Checking Equivalence for Reo Networks. *Electronic Notes in Theoretical Computer Science*, 215:209–226, 2008.

[BBKK09a] C Baier, T Blechmann, J Klein, and S Klüppelholz. Formal verification for components and connectors. In *Formal Methods for Components and Objects*, volume 5751, pages 82–101. Springer, 2009.

[BBKK09b] C Baier, T Blechmann, J Klein, and S Klüppelholz. A uniform framework for modeling and verifying components and connectors. In *Coordination Models and Languages*, volume 5521 of *Lecture Notes in Computer Science*, pages 247–267. Springer, 2009.

[BCG88] M C Browne, E M Clarke, and O Grümberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1–2):115 – 131, 1988.

[BCM+92] J R Burch, E M Clarke, K L McMillan, D L Dill, and L J Hwang. Symbolic model checking: 1020 States and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BDE+12] C Baier, M Daum, B Engel, H Härtig, J Klein, S Klüppelholz, S Märcker, H Tews, and M Völp. Chiefly symmetric: Results on the scalability of probabilistic model checking for operating-system code. In Proceedings Seventh Conference on *Systems Software Verification* ,Sydney, Australia, 28-30 November 2012, volume 102 of *Electronic Proceedings in Theoretical Computer Science*, pages 156–166. Open Publishing Association, 2012.

[BK08] C Baier and J P Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BKKa] T Blechmann, J Klein, and S Klüppelholz. Vereofy user manual, http://www.vereofy.de/.

[BKKb] T Blechmann, J Klein, and S Klüppelholz. Vereofy website. http://www.vereofy.de/. Last accessed: 09/09/2012, 12:45 PM.

[BKK11] C Baier, J Klein, and S Klüppelholz. Modeling and verification of components and connectors. In *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 114–147. Springer, 2011.

[BP02] S Blom and J Pol. State space reduction by proving confluence. In *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 596–609. Springer, 2002.

[Bry86] R E Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677 –691, 1986.

[BSAR06] C Baier, M Sirjani, F Arbab, and J Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.

[CE82] E M Clarke and E A Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1982.

[CEFJ96] E M Clarke, R Enders, T Filkorn, and S Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2):77–104, 1996.

[CEJS98] E M Clarke, E A Emerson, S Jha, and A P Sistla. *Symmetry reductions in model checking*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.

[CGJ+01] E Clarke, O Grumberg, S Jha, Y Lu, and H Veith. Progress on the state explosion problem in model checking. In *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 176–194. Springer, 2001.

[CJ95] E M Clarke and S Jha. Symmetry and induction in model checking. In *Computer Science Today*, pages 455–470. Springer, 1995.

[CK96] E M Clarke and R P Kurshan. Computer-aided verification. *Spectrum*, 33(6):61–67, 1996.

[CMCHG96] E Clarke, K McMillan, S Campos, and V Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification*, pages 419–422. Springer, 1996.

[CW96] E M Clarke and J M Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[DHH+06] M B Dwyer, J Hatcliff, M Hoosier, V R, Robby, and T Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 73–89. Springer, 2006.

[Dij78] E W Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. In *Programming Methodology*, pages 166–175. Springer, 1978.

[DM05] A F Donaldson and A Miller. Automatic symmetry detection for model checking using computational group theory. In *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 481–496. Springer, 2005.

[DM06] A F Donaldson and A Miller. Symmetry reduction for probabilistic model checking using generic representatives. In *Automated Technology for Verification and Analysis*, volume 4218 of *Lecture Notes in Computer Science*, pages 9–23. Springer, 2006.

[DMP07] A F Donaldson, A Miller, and D Parker. Grip: Generic representatives in prism. In *Quantitative Evaluation of Systems, 2007. QEST 2007. Fourth International Conference on the*, pages 115–116, 2007.

[ES96] E A Emerson and A P Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9:105–131, 1996.

[ES97] E A Emerson and A P Sistla. Utilizing symmetry when model-checking under fairness assumptions: an automata-theoretic approach. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, 1997.

[ET03] E A Emerson and R J Trefler. From Asymmetry to Full Symmetry: New Techniques for Symmetry Reduction in Model Checking. In *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2003.

[EW03] E A Emerson and T Wahl. On Combining Symmetry Reduction and Symbolic Representation for Efficient Model Checking. In *Lecture Notes in Computer Science*, pages 216–230. Springer, 2003.

[GP93] P Godefroid and D Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449. Springer, 1993.

[GS05] V Gyuris and A P Sistla. On-the-fly model checking under fairness that exploits symmetry. In *Computer Aided Verification*, pages 232–243. Springer, 2005.

[GvLH+96] P Godefroid, J van Leeuwen, J Hartmanis, G Goos, and P Wolper. Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem, 1996.

[HDZ00] J Hatcliff, M B Dwyer, and H Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[HP94] G J Holzmann and D Peled. An improvement in formal verification. In *Proceedings of the 7th IFIP WG6*, volume 1, pages 197–211, 1994.

[ID96] C N IP and D L Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1):41–75, 1996.

[Ios02] R Iosif. Symmetry reduction criteria for software model checking. In *Model Checking Software*, pages 22–41. Springer, 2002.

[KB07] S Klüppelholz and C Baier. Symbolic Model Checking for Channel-based Component Connectors. *Electronic Notes in Theoretical Computer Science*, 175(2):19–37, 2007.

[Klü12] S Klüppelholz. *Verification of Branching-Time and Alternating-Time Properties for Exogenous Coordination Models*. PhD thesis, Technische Universität Dresden, 2012.

[KLY02] R Kurshan, V Levin, and H Yenigün. Compressing transitions for model checking. In *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 569–582. Springer, 2002.

[KNP06] M Kwiatkowska, G Norman, and D Parker. Symmetry reduction for probabilistic model checking. In *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 2006.

[Lip75] R J Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[Maz87] A Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *Lecture Notes in Computer Science*, pages 278–324. Springer, 1987.

[McM92] K L McMillan. *Symbolic Model Checking: An approach to the state explosion problem*. PhD thesis, Carnegie Mellon University (CMU), Carnegie Melon University, 1992.

[MDC06] A Miller, A Donaldson, and M Calder. Symmetry in temporal logic model checking. *ACM Computing Surveys*, 38(3), 2006.

[Mer01] S Merz. Model checking: A tutorial overview. In *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer, 2001.

[Mil80] R Milner. *A calculus of communicating systems*. Lecture notes in computer science. Springer, 1980.

[Mil12] J S Milne. Group theory (v3.12), 2012. Available at http://www.jmilne.org/math/.

[MVO04] C Menini and F Van Oystaeyen. *Abstract algebra*, volume 263 of *Monographs and Textbooks in Pure and Applied Mathematics*. Marcel Dekker Inc., 2004.

[Par81] D Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[Pel93] D Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993.

[Pel96] D Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996.

[Pel09] R Pelánek. Fighting state space explosion: Review and evaluation. In *Formal Methods for Industrial Critical Systems*, volume 5596 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 2009.

[QS82] J P Queille and J Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[SG04] A P Sistla and P Godefroid. Symmetry and reduced symmetry in model checking. *ACM Transactions on Programming Languages and Systems*, 26(4):702–734, 2004.

[Tip95] F Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[Val91a] A Valmari. A stubborn attack on state explosion. In E M Clarke and R P Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165. Springer, 1991.

[Val91b] A Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1991.

[vG90] R J van Glabbeek. The linear time – branching time spectrum. In *CONCUR '90 Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 1990.

[vG93] R J van Glabbeek. The linear time – Branching time spectrum II. In *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993.

[Wei84] M Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[Wol95] P Wolper. An introduction to model checking. *Position statement for panel discussion at the Software Quality workshop*, 1995.

A The Code of Symmetry Library

```
#include "builtin"

MODULE ACTIVITY_CHECKER_2{
  in: A;
  in: B;

  out: C;

  -[ IF(A | B) & C]->;
}

CIRCUIT ACTIVITY_CHECKER<k>{

  if(k==1){
    sync = new SYNC(source[0];sink[0]);
  }

  if(k>1){
    AC[0] = new ACTIVITY_CHECKER_2(source[0],source[1];N[0]);

    for(i=2; i < k; i=i+1){
      AC[i-1] = new ACTIVITY_CHECKER_2(N[i-2],source[i];N[i-1]);
    }
    sink[0] = N[k-2];
  }
}

CIRCUIT sym_scenario_1<k,T> {
  excluder = new READER_INF(EXNODE);

  if(k==0){
    ERROR_k0 = new SYNC;
  }
```

```
  for (i=0;i<k;i=i+1) {
    c[i]=new T;

    if(c[i].sources + c[i].sinks < 1 ){
       if(i==0){
         ERROR = new SYNC;
       }
    }
    else{
     if(i==0){
       for(j=0;j<c[i].sources;j=j+1){
         source[j] = ROUTE_NODE;
       }
       for(j=0;j<c[i].sinks;j=j+1){
         sink[j] = NODE;
       }
     }

     ac[i] = new ACTIVITY_CHECKER<c[i].sources + c[i].sinks>;
     join(EXNODE,ac[i].sink);

     if(c[i].sources > 0){
       for(j=0;j<c[i].sources;j=j+1){
         inMEDNODE[i*c[i].sources+j] =
           join(c[i].source[j],ac[i].source[j]);
         insyncs[i*c[i].sources+j] =
            new SYNC(source[j];inMEDNODE[i*c[i].sources+j]);
       }
     }
     if(c[i].sinks > 0){
       for(j=0;j<c[i].sinks;j=j+1){
         outMEDNODE[i*c[i].sinks+j] =
           join(c[i].sink[j],ac[i].source[(c[i].sources) + j]);
         outsyncs[i*c[i].sinks+j] =
           new SYNC(outMEDNODE[i*c[i].sinks+j];sink[j]);
       }
     }
   }
  }
}


CIRCUIT sym_scenario_2<k,T> {
  inexcluder = new READER_INF(INEXNODE);
  outexcluder = new READER_INF(OUTEXNODE);

  if(k==0){
    ERROR_k0 = new SYNC;
  }
```

```
  for (i=0;i<k;i=i+1) {
    c[i]=new T;

    if(c[i].sources + c[i].sinks < 1 ){
       if(i==0){
         ERROR = new SYNC;
       }
    }
    else{
     if(i==0){
       for(j=0;j<c[i].sources;j=j+1){
         source[j] = ROUTE_NODE;
       }
       for(j=0;j<c[i].sinks;j=j+1){
         sink[j] = NODE;
       }
     }

     if(c[i].sources > 0){
       inac[i] = new ACTIVITY_CHECKER<c[i].sources>;
       join(INEXNODE,inac[i].sink);
       for(j=0;j<c[i].sources;j=j+1){
         inMEDNODE[i*c[i].sources+j] =
           join(c[i].source[j],inac[i].source[j]);
         insyncs[i*c[i].sources+j] =
           new SYNC(source[j];inMEDNODE[i*c[i].sources+j]);
       }
     }
     if(c[i].sinks > 0){
       outac[i] = new ACTIVITY_CHECKER<c[i].sinks>;
       join(OUTEXNODE,outac[i].sink);
       for(j=0;j<c[i].sinks;j=j+1){
         outMEDNODE[i*c[i].sinks+j] =
           join(c[i].sink[j],outac[i].source[j]);
         outsyncs[i*c[i].sinks+j] =
           new SYNC(outMEDNODE[i*c[i].sinks+j];sink[j]);
       }
     }
   }
  }
}

CIRCUIT sym_scenario_3<k,T> {
  excluder = new READER_INF(EXNODE);

  if(k==0){
    ERROR_k0 = new SYNC;
  }
```

```
  for (i=0;i<k;i=i+1) {
    c[i]=new T;

    if(c[i].sources + c[i].sinks < 1 ){
       if(i==0){
         ERROR = new SYNC;
       }
    }
    else{
     if(i==0){
       for(j=0;j<c[i].sources;j=j+1){
         source[j] = ROUTE_NODE;
       }
       for(j=0;j<c[i].sinks;j=j+1){
         sink[j] = NODE;
       }
     }

     ac[i] = new ACTIVITY_CHECKER<c[i].sources + c[i].sinks>;
     join(EXNODE,ac[i].sink);

     if(c[i].sources > 0){
       for(j=0;j<c[i].sources;j=j+1){
         inMEDNODE[i*c[i].sources+j] =
           join(c[i].source[j],ac[i].source[j]);
         insyncs[i*c[i].sources+j] =
           new SYNC(source[j];inMEDNODE[i*c[i].sources+j]);
       }
     }
     if(c[i].sinks > 0){
       for(j=0;j<c[i].sinks;j=j+1){
         outMEDNODE[i*c[i].sinks+j] =
           join(c[i].sink[j],ac[i].source[(c[i].sources) + j]);
         outsyncs[i*c[i].sinks+j] =
           new SYNC(outMEDNODE[i*c[i].sinks+j];sink[j]);
       }
     }
   }
  }
}

CIRCUIT sym_scenario_4<k,T> {
  inexcluder = new READER_INF(INEXNODE);
  outexcluder = new READER_INF(OUTEXNODE);

  if(k==0){
    ERROR_k0 = new SYNC;
  }
```

```
  for (i=0;i<k;i=i+1) {
    c[i]=new T;

    if(c[i].sources + c[i].sinks < 1 ){
       if(i==0){
         ERROR = new SYNC;
       }
    }
    else{
     if(i==0){
       for(j=0;j<c[i].sources;j=j+1){
         source[j] = ROUTE_NODE;
       }
       for(j=0;j<c[i].sinks;j=j+1){
         sink[j] = NODE;
       }
     }

     if(c[i].sources > 0){
       inac[i] = new ACTIVITY_CHECKER<c[i].sources>;
       join(INEXNODE,inac[i].sink);
       for(j=0;j<c[i].sources;j=j+1){
         inMEDNODE[i*c[i].sources+j] =
           join(c[i].source[j],inac[i].source[j]);
         insyncs[i*c[i].sources+j] =
           new SYNC(source[j];inMEDNODE[i*c[i].sources+j]);
       }
     }
     if(c[i].sinks > 0){
       outac[i] = new ACTIVITY_CHECKER<c[i].sinks>;
       join(OUTEXNODE,outac[i].sink);
       for(j=0;j<c[i].sinks;j=j+1){
         outMEDNODE[i*c[i].sinks+j] =
           join(c[i].sink[j],outac[i].source[j]);
         outsyncs[i*c[i].sinks+j] =
           new SYNC(outMEDNODE[i*c[i].sinks+j];sink[j]);
       }
     }
   }
  }
}
```