UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND APPLIED SCIENCE

Department of Electronics and Computer Science

# Improved State-Space Construction in Automated Verification

by

**Simon P. St James**

Thesis for the degree of Doctor of Philosophy

~~September 2003~~
May 2008

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

IMPROVED STATE-SPACE CONSTRUCTION IN AUTOMATED VERIFICATION

by Simon P. St James

Verifying system specifications using traditional model-checking techniques rapidly becomes infeasible as the complexity of the specification becomes non-trivial, due to the state-space explosion problem, wherein the representation of the behaviour of the system becomes too large to be practically constructable. Thus, we require techniques that collapse the state-space to a manageable size while still preserving the information required for verification of the desired properties.

The concept of abstraction provides one effective means of combatting state-space explosion. Essentially, abstraction aims to simplify the behaviour by hiding details that are not directly relevant to the verification task. Within the abstraction framework, the original behaviour of the system is known as the concrete behaviour, and the simplified behaviour the abstract. The precise means of abstraction we consider acts by reducing the set of actions appearing in the abstract behaviour by means of a mapping from each of the set of concrete actions to an abstract action (action renaming) or to the empty word (action hiding). It has been previously shown that when the abstraction fulfils a condition called weak continuation-closure, then the abstract behaviour can be used to decide whether or not the concrete behaviour satisfies a property under a satisfaction relation called satisfaction within fairness, a relation that includes a built-in concept of fairness. The drawback is that the technique requires the construction of the original state-space, which is often infeasible.

The main contribution of this thesis is to show that partial-order reduction can be combined with abstraction in such a way that the the abstraction can be used to decide whether the concrete behaviour satisfies a given property within fairness using only a partial-order reduced version of the state-space, which potentially could be orders of magnitude smaller than the full state-space. Attention is also paid to providing practical means for computing this partial-order reduction, and a couple of results in the field of compositional verification are presented.

# Contents

# List of Figures

# List of Tables

# Acknowledgements

Thanks go foremost to my supervisor Uli, who posed interesting problems and was always patient with me and was always there to keep me motivated. During my PhD, I have come to regard him as a very good friend.

Part of the work on this thesis was funded by EPSRC (grant number GR/N23684), which I very gratefully acknowledge.

# Chapter 1

# Introduction

## 1.1 Automated Verification

*Automated verification*, in the sense used throughout this thesis, is the mechanical testing of a system specification to check whether it contains errors and conforms to a required *property.* The most naïve means of doing this is to build the *state-space* of the system: a labelled transition system containing all possible states the system may move through as it executes all of its possible *computations.* A *partial-computation* is a finite sequence of *actions*, each of which is viewed as being an atomic event within the system; a *computation* of a system is an infinitely extended partial computation. The *behaviour* of the system is defined as the set of all possible computations of the system; thus the state-space of the system, once constructed, affords a relatively easily exploited encoding of the behaviour of the system.

## 1.2 Properties and Satisfaction

Verification of a system specification often equates to verifying that the behaviour of a system "satisfies" a property. In this thesis, properties are identified with the set of all "correct" computations and are typically formulated as a propositional linear-time temporal logic formula [Pnueli (1977)]. When deciding whether a system specification "satisfies" a property, some measure of "fairness" is usually introduced into the satisfaction condition in order to filter out those computations of the system that do not conform to the property but which, while technically possible within the system, are viewed as "unfair" or unrealistic sequences of events e.g. a computation in which an action that is required to be taken in order for the computation to satisfy the property has infinitely many chances to be taken, but never is. The definition of satisfaction we will deal with is known as "satisfaction within fairness"[1][Nitsche and Ochsenschläger (1996); Nitsche and Wolper (1997)]. In essence, this definition says that a system behaviour satisfies a property within fairness if and only if every partial computation of a system can be extended to an infinite one that satisfies the property .

## 1.3 State-space Explosion

The aforementioned state-space of the system specification often grows exponentially with the complexity of the system specification, a phenomenon known as the *state-space explosion* problem; ameliorating this problem while verifying that the system specification satisfies properties within fairness forms the main focus of this thesis. The main result of this thesis combines two different means of state-space reduction: *abstraction* and *partial-order reduction.*

## 1.4 Abstraction

Abstraction aims to reduce the size of the state-space by effectively "simplifying" the system specification being considered by "hiding" some of the internal details of how system components work. For example, if we had a system specification detailing a communication protocol, we may be more interested in verifying whether a message sent via the protocol is received by its intended recipient; the underlying mechanisms employed by the protocol may be of no interest in the verification procedure, and only the emergent behaviour (whether the sent message reaches its intended recipient) need be verified. In this case, substantial reduction in the state-space may be obtained by "abstracting away" these implementation details and only exposing the higher-level behaviours to the verification process.

The method of abstraction employed throughout this thesis is that of *behaviour abstraction*, wherein groups of actions are either mapped to some action which is viewed as important with respect to the property we wish to verify (*action renaming*) or, more usefully, mapped to the empty word if the actions are not of immediate importance to the property being verified. This mapping on the set of actions is called an *abstracting homomorphism.* In the previously mentioned example, we may wish to map all actions comprising the low-level workings of the protocol to the empty word, and map the action that initiates a "send" event to some action (*msg_send*, say) and the action that signals a "receive" event to another action (*msg_receive*, say). The resulting behaviour, with this abstracting homomorphism applied, will be very much simpler than the full, detailed behaviour, and the state-space will be correspondingly smaller.

[1]Originally, "satisfaction up to liveness".

This approach is not too useful unless we may infer properties satisfied by the system behaviour based on the abstract behaviour, and in fact it was shown in [Nitsche and Ochsenschläger (1996)] that a specific class of abstracting homomorphisms (called *weakly continuation-closed* homomorphisms) preserve satisfaction within fairness of properties; that is, if the applied abstracting homomorphism is weakly continuation closed on the full behaviour, the resulting abstract behaviour will satisfy a given propery within fairness if and only if the full behaviour satisfies that property within fairness.

However, the usefulness of this result is diminished by the fact that computing this abstract behaviour still required one to compute the *full* behaviour (state-space) of the system at some point. The main contribution of this thesis is proving that we may avoid having to construct this full state-space in favour of a *partial-order reduced* version.

## 1.5 Partial-order Reduction

Another means for reducing the size of a state-space stems from the observation that some pairs of actions are "independent" of one another, in the sense that neither interferes with the other and that following the pair of actions in either order leads to the same state. Partial computations that may be transformed into one another by swapping the order of independent actions adjacent to one another in the sequence, then, are in a limited sense functionally equivalent, and the standard state-space construction will explore all such "redundant" computations. The *persistent set selective search* techniques of Godefroid and Wolper [Godefroid (1991, 1995); Godefroid and Wolper (1993); Wolper and Godefroid (1991)] aim to reduce the state-space by avoiding following some of these redundant interleavings. The primary contribution of this thesis is to show that these persistent set selective search techniques, in particular those which yield what is known as a *trace reduction* of the full system behaviour, can, when modified to be "compatible" with the abstracting homomorphism, give a reduced representation of the full behaviour that can completely take the place of the full state-space required earlier. Thus, we may use a combination of partial order reduction and abstraction to infer the properties satisfied within fairness by the full behaviour without ever having to construct the state-space representing it.

## 1.6 Thesis Statement

The current requirement of constructing the entire state-space before performing verification on the smaller resulting abstract representation negates many of the benefits of using abstraction in the first place and is in fact unnecessary: It is possible to use a partial-order reduced version of the state-space instead, which can be significantly smaller.

## 1.7 Organisation of this Thesis

Chapter 2 deals with the formal definitions of the concepts mentioned in this Introduction. Chapter 3 deals with the main result of this thesis: proving that a partial-order reduction of the state-space, appropriately modified to be compatible with the abstracting homomorphism, can be used to decide which properties are satisfied within fairness by the full behaviour. Chapter 4 provides a formalism of *high-level Petri nets* for specifying systems (Petri nets are chosen as they are generally very amenable to partial-order reduction methods) and aims to give a practical means for computing the persistent sets required by the persistent set selective search by examining existing persistent set construction algorithms and exploring how these might need to be adapted to work more optimally with the special requirements of the result of Chapter 3. Chapter 5 presents a result that combines partial-order reduction and abstraction with compositional verification. Chapter 6 gives an experimental justification of the main result of this thesis by testing the the modified persistent set algorithm on a practical example. We conclude the main thesis by giving a summary of the results presented and a comparison with existing work, and suggesting some avenues for future research.

The five Appendices to the thesis contain material which, for various reasons, is largely of minor importance or of purely theoretical interest. Appendix A describes an algorithm for deciding whether a given abstracting homomorphism is weakly-continuation closed on a given behaviour; in the case where the efficiently checkable sufficient conditions are of no help, this result is more efficient than the existing technique. Appendix B contains a variant of the persistent set selective search optimised for the case where we wish only to extract the abstract behaviour of a system specification, as is the case in the result of Chapter 5. Appendix C answers an open question concerning whether a result called "the commuting limits theorem" holds for non-regular languages. Appendix D gives an explicit algorithm for deciding whether a language satisfies a property within fairness. Finally, Appendix E gives a comparison of some definitions of "fairness" of a satisfaction relation, and shows where our notion of *satisfaction within fairness* fits in the hierarchy.

# Chapter 2

# Preliminaries

## 2.1 Overview

This chapter introduces most of the core concepts used in this thesis, starting with some Automata and Language Theory. We then move on to model checking and describe some existing techniques to reduce the size of the state space during the model checking process (the techniques of *abstraction* and *partial-order reduction*) and describe what it means for a system to model a property under different notions of "fairness".

## 2.2 Automata and languages

In this section, we will present some of the required definitions from automata and language theory [Hopcroft and Ullman (1979); Harrison (1978)].

**Definition 2.1.** Formally, a *finite automaton* $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ consists of a finite set $Q$ of *states*, an *input alphabet* $\Sigma$, a *transition relation* $\delta \subseteq Q \times \Sigma \times Q$, an *initial state* $q_0 \in Q$, and a set $F$ of *accepting states.*

In all examples we consider in this document, the set $F$ is trivially set to $Q$; i.e. all states are accepting.

**Definition 2.2.** When an automaton $\mathcal{A}$ satisfies the condition $F = Q$, it is called a *labelled transition system*, or an *LTS* for short. We write $\mathcal{A} = (Q, \Sigma, \delta, q_0)$ if $\mathcal{A}$ is an LTS.

Every automaton of this kind encodes a language over the alphabet $\Sigma$, as described in the next few definitions.

**Definition 2.3.** A *run* of $\mathcal{A}$ on a finite word $x = x_1x_2...x_n \in \Sigma^*$ ($x_i \in \Sigma$) is a sequence

$$q_0q_1...q_n$$

$[q_i \in Q]$ such that $(q_{i-1}, x_i, q_i) \in \delta$ for each $i = 1, 2, ..., n$. The automaton $\mathcal{A}$ *accepts* this word $x$ if and only if $q_n \in F$.

Since, for an LTS, $F = Q$, the condition for acceptance of $x$ by an LTS $\mathcal{A}$ is simply that there exists a run of $\mathcal{A}$ on $x$.

**Definition 2.4.** Let $L$ be the set of words accepted by $\mathcal{A}$ ; i.e.

$L = \{x \in \Sigma^* | \mathcal{A} \text{ accepts } x\}$.

Then we call $L$ the *language accepted by* $\mathcal{A}$.

It is also possible to define the acceptance by $\mathcal{A}$ of *infinite* words (or $\omega$*−words*), a notion usually called Büchi-acceptance[Büchi (1962)], as we will begin to do now.

**Definition 2.5.** A run of $\mathcal{A}$ on an $\omega-$word $x = x_1x_2... \in \Sigma^\omega$ ($x_i \in \Sigma$) is an infinite sequence

$$q_0q_1q_2...$$

$[q_i \in Q]$ such that $(q_{i-1}, x_i, q_i) \in \delta$ for all $i = 1, 2, 3, ...$

For a run $r \in Q^\omega$, define $\omega(r)$ to be the set of states occurring infinitely many times in $r$; we say then that:

**Definition 2.6.** The automaton $\mathcal{A}$ Büchi–accepts the $\omega-$word $x = x_1x_2... \in \Sigma^\omega$ if and only if there is a run $r$ of $x$ satisfying $\omega(r) \cap F \neq \phi$.

Obviously, in the case where $\mathcal{A}$ is an LTS, this condition again simplifies to $\mathcal{A}$ Büchi-accepts $x$ if and only if there is a run of $\mathcal{A}$ on $x$. As before, the language $L_\omega$ Büchi-accepted by the automaton $\mathcal{A}$ is simply the set of infinite words Büchi-accepted by $\mathcal{A}$ (a language consisting of $\omega-$words, such as $L_\omega$, is called an $\omega$*−language*). We will have a little more to say about the language Büchi-accepted by an LTS $\mathcal{A}$ after the following important set of definitions.

**Definition 2.7.** Let $x = x_1x_2....x_n \in \Sigma^*$ be a word over $\Sigma$. Then we define:

$$pre(x) = \{x_1x_2...x_m | 0 \leq m \leq n\}$$

i.e. the set of words that are the "beginning segments" of $x$. Similarly, we define for an $\omega-$word $x = x_1x_2.... \in \Sigma^\omega$

$$pre(x) = \{x_1x_2...x_m | m \geq 0\}$$

In either case, if $y \in pre(x)$, we write $y \preceq x$. If $y \in pre(x)$ but $y \neq x$, we write $y \prec x$. Note that the $m = 0$ case corresponds to the empty word $\varepsilon$. We extend the definition to languages in the obvious way:-

$$pre(L) = \{y \in L | \exists x \in L : y \preceq x\}$$

where $L$ may be either an ordinary language or an $\omega-$language.

**Definition 2.8.** A language $L$ is said to be prefix-closed if and only if $pre(L) = L$; that is, if a word $w$ is in $L$, then all prefixes of $w$ are in $L$.

Note that the language accepted by an LTS is always prefix-closed.

**Definition 2.9.** We define the *Eilenberg Limit* [Eilenberg (1974)] of the language $L \subseteq \Sigma^*$ to be the $\omega-$language over $\Sigma$ defined as

$$\lim(L) = \{x \in \Sigma^\omega | \exists^\infty w \in pre(x) : w \in L\}$$

where "$\exists^\infty$..." should be read as "there exist infinitely many different...".

Perhaps a more intuitive definition would be to say that a word $x \in \Sigma^\omega$ is in $\lim(L)$ if and only if there is a sequence $w_1, w_2, ...$ of words of increasing length in $L$ such that $w_i \prec w_{i+1}$ for each $i = 1, 2, ...$ and $w_i \prec x$ for all $i = 1, 2, ....$ Thus, an $\omega$-word $x$ is in $\lim(L)$ if and only if there is a sequence of finite words, all contained in $L$, each extending the previous word by appending actions to it and, very informally, "converging" to the $\omega$-word $x$ as this process is continued to infinity.

Note that in the case where $L$ is prefix-closed, Definition 2.9 may be simplified to

$$\lim(L) = \{x \in \Sigma^\omega | \forall w \in pre(x) : w \in L\}$$

as follows. Let $L \subseteq \Sigma^*$ be a prefix-closed language and let $x \in \Sigma^\omega$ be an $\omega$-word in $\lim(L)$. Then as noted, we would have a sequence $w_1, w_2, ...$ of words in $L$ such that $w_i \prec w_{i+1}$ for each $i = 1, 2, ...$ and $w_i \prec x$ for all $i = 1, 2, ....$ Given any word $w \in L$ where $w \prec x$, then $w$ would be a prefix of all words $w_i$ for which $|w_i| >= |w|$ and, since all $w_i$ are in $L$ and $L$ is prefix-closed, $w$ would also be in $L$.

In the case where $\mathcal{A}$ is an LTS, it can be shown that the language Büchi-accepted by $\mathcal{A}$ coincides with the Eilenberg Limit of the finitary language accepted by $\mathcal{A}$ [Thomas (1990)] , so obtaining a representation of $\lim(L)$ given a representation of $L$ is trivial.

**Definition 2.10.** The *leftquotient* [Harrison (1978)] or *continuation* of a word $w \in \Sigma^*$ in a language $L \subseteq \Sigma^*$, denoted $cont(w, L)$, is defined as

$$cont(w, L) = \{y \in \Sigma^* | wy \in L\}$$

with a similar definition for $\omega$−languages $L_\omega \subseteq \Sigma^\omega$;

$$cont(w, L_\omega) = \{y \in \Sigma^\omega | wy \in L_\omega\}$$

**Note 2.1** It is an important fact that the elements in the set $\{cont(w, L) | w \in L\}$ correspond exactly to the states in the minimal automaton representing the language $L$. To make this correspondence clear, take any state $s$ in this minimal automaton and find all words accepted from that state (i.e. this operation is equivalent to temporarily switching the initial state of the automaton to $s$, and finding the language accepted by this new automaton); then this will be one of the sets in $\{cont(w, L) | w \in L\}$. In fact, it will be equal to the set $cont(w, L)$, where $w$ is any word that leads from the original initial state to $s$; thus, as an aside, we see that the sets $cont(w, L)$ and $cont(w', L)$ (with $w, w' \in \Sigma^*$) are equal if and only if the words $w$ and $w'$ lead to the same state in the minimal automaton representing $L$. As a consequence, most definitions involving a state $s$ in this thesis are implicitly extended to any word $w$ leading from $s_0$ to $s$, and most definitions in terms of a word $w$ in a language are extended to definitions in terms of the state $s$ reached by following $w$ from $s_0$ in the automaton representing that language.

For example, we extend Definition 2.10 to a definition involving states as follows: let $\mathcal{A}$ be the automaton representing $L$, and let $s$ be a state of $\mathcal{A}$ that can be reached in $\mathcal{A}$ by following a word $w \in L$. Then we define

$$cont(s, L) = cont(w, L)$$

As mentioned, the language represented by $cont(s, L)$ is precisely the language accepted by the automaton obtained by changing the initial state of $\mathcal{A}$ from $q_0$ to $s$, which provides an efficient means for computing $cont(s, L)$ and, by extension, $cont(w, L)$.

**Definition 2.11.** A word $w \in L$ is *maximal* in $L$ if $cont(w, L) = \{\varepsilon\}$; i.e. no further actions may be followed after following $w$.

The state in the aforementioned minimal automaton corresponding to such a maximal word would be a *dead-end* or a *deadlock* state, from which no further progress is possible. We should make an important note at this point; the main result of this thesis is not directly compatible with automata containing deadlocks (nor, by implication, languages containing maximal words) - more precisely, Theorem 2.18 requires, for technical reasons, that the automaton not contain deadlocks. To overcome this restriction, we do the following: we introduce a "dummy" action, which we will denote by "#". If an automaton contains a deadlock state, then we introduce a transition from that state to itself, labelled with "#" (so the formerly maximal words in the corresponding language now have the set $\{\#^n | n \geq 0\}$ as their continuation). The input alphabet is now $\Sigma \cup \{\#\}$;

to avoid this cumbersome notation, however, we will simply assume that the "#" action has been included and just write $\Sigma$ instead.

Strongly-connected components[Aho et al. (1974)] are used in several places in this thesis, and we introduce them here.

**Definition 2.12.** Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be an automaton. Then a *strongly connected component* $C \subseteq Q$ of $\mathcal{A}$ is a subset of the set of states of $\mathcal{A}$ with the property that, for any pair $s, s' \in C$ of states in $C$, there is a path

$$s = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 ... s_n \xrightarrow{a_n} s_{n+1} = s'$$

such that $s_i \in C$ for all $i = 1, 2, ...n + 1$ and $(s_i, a_i, s_{i+1}) \in \delta$ for all $i = 1, 2, ...n$.

An automaton $\mathcal{A}$ is *strongly-connected* if all of its states form a single, strongly connected component. If $C \subseteq Q$ is a strongly-connected component of $\mathcal{A}$ and there is no $C' \subseteq Q$ such that $C \subset C'$ and $C'$ is also a strongly-connected component of $\mathcal{A}$, then $C'$ is said to be *maximal.*

A *strongly connected bottom component* $C$ of $\mathcal{A}$ is a strongly connected component of $\mathcal{A}$ with the additional property that there does not exist a transition $(s, a, s') \in \delta$ with $s \in C$ and $s' \notin C$; it represents a strongly connected component with no transitions leading out of the strongly connected component.

It is a fact that from any state $s$ in any automaton $\mathcal{A}$, we may always reach from $s$ either a deadlock state or a strongly connected bottom component.

Having dealt with the automata-related preliminaries, we now move on to the idea behind model-checking.

## 2.3 Model-Checking

*Model-checking*, loosely speaking, consists of modelling the behaviour of a real-life system within a computer (by means of an appropriate *system specification*) in order to check whether the behaviour is a *model* of some property $\mathcal{P}$, as expressed in some logical language.

### 2.3.1 System Specifications

For our purposes, a specification of a system must contain the following information:

A set of *actions*: Intuitively, this is the set of elementary operations that the system may perform, or events that may occur within the system; for example, sending or receiving messages, dealing with a client in a queue, etc.

A notion of the *states* of the system: A *state* may be likened loosely to a snap-shot of the system at a given moment in time, and should contain all information describing the internal configuration of the system at that moment to a level of detail deemed sufficient for the verification of the chosen properties[1]. For example, the state of a system could contain the list of the clients in the queues, the progress of a message through a communication network, the values of variables at the current stage of a calculation, etc.

A *transition relation*: This tells one which state the system enters after performing action $a$ at state $s$; i.e. it contains the information describing the evolution of the system during a sequence of actions. There should also be a rule that allows one to tell which actions the system is immediately able to perform at each state $s$, based on the information contained in the internal representation of $s$; this set is called the set of *enabled actions* at the state $s$, denoted $En(s)$. By Note 2.1, we extend this definition to $En(w)$ (where $w$ is any word such that following $w$ from $s_0$ leads to $s$), "the set of enabled actions at $w$".

Given this information, we may form an automaton representing the 'behaviour' of the system (the inverted commas are used because the *behaviour* of a system has a more formal definition, given later - see Definition 2.14) in an obvious way: the set of states $Q$ is the set of states of the system reachable from the *initial state* of the system; the set $\Sigma$ is the set of actions of the system, and $\delta$ is identified with the transition relation of the system.

We form this automaton $\mathcal{A}$ by starting with the initial state $s_0$ of the system, finding all enabled actions $En(s_0)$ at this state, firing each in turn and using the transition relation to see which new states are entered. Then for each of these new states, we find the set of enabled actions, fire them, work out which new states are entered, etc. We carry on until no new states are created. More formally, the procedure may be described as in Algorithm 2.1. Note that the particular formulation given in Algorithm 2.1 is that of a breadth-first search, but this is not a necessity and a depth-first version would work equally well.

The resulting set of states in this LTS is called the *state-space* of the system. Let $L$ henceforth denote the language accepted by $\mathcal{A}$.

**Definition 2.13.** A *partial computation* of the system is a word in $L$. A *computation*

[1] Obviously, one may increase the detail of the model to an arbitrary degree, but there of course comes a point where one must assume that the model is sufficiently detailed to justify the inference of a property of the real-life system from a property of the model. This notion of *sufficient detail* is quite closely linked with that of *abstraction*, which we will describe later.

```
$Q \leftarrow \phi$;$NewStates \leftarrow s_0$;
do while $NewStates \neq \phi$
    pick $s \in NewStates$;
    $NewStates \leftarrow NewStates - s$;
    $Q \leftarrow Q \cup s$;
    let $T = En(s)$;
    if $T = \phi$ then              // The state $s$ must be a deadlock
        $\delta \leftarrow \delta \cup (s, \#, s)$;  // state. Add a loop to itself marked
    endif                           // with the dummy label, "#"
    for each action $a$ in $T$
        for each states $s'$ such that firing $a$ at $s$ leads to $s'$
            if $s' \notin Q$
                $NewStates \leftarrow NewStates \cup s'$;
            endif
        next
        $\delta \leftarrow \delta \cup (s, a, s')$;
    next
loop
```

**Algorithm 2.1** – Basic State-Space Exploration Algorithm for Constructing the LTS Representing the System

is an infinitely extended partial computation; i.e. a word in $\lim(L)$.

**Definition 2.14.** The *behaviour* of the system is the set of all these computations; i.e is equal to $\lim(L)$.

The usage of *infinite* sequences of actions need some justification: Even though protocols are finite, usually one considers consecutive runs of them, leading to infinite system behaviour. In the case where one is dealing with terminating systems, one can add to each potential termination (deadlock) state a transition to a new state with a self-loop labelled with special symbol standing for "I have terminated"; as mentioned earlier, we use "#" for this special symbol in this thesis.

#### 2.3.1.1 Introduction to Petri Nets

One method for specifying systems that is particularly amenable to the partial-order methods we will employ in this thesis is to use *Petri Nets*[Jensen (1997); Hack (1976); Reisig (1985)]. There are various 'levels' of Petri Nets, the simplest (and least useful practically) being *low-level Petri Nets* . These consist of a numbered set of *places*, each of which contains a number of *tokens*, and a set of *actions* $(\Sigma)^2$. Places are represented

[2] In most of the literature on Petri Nets, this set is called the set of *transitions*; we have adopted an alternative notation to emphasize the fact that this set corresponds to the set of elementary operations

by circles with the number of tokens contained in the centre, and the place-number located on the outside[3]; actions are represented by rectangles with the action name inside. Each action may have a number of arrows pointing into places, and a number of arrows pointing in to it from places. Information of the former kind is encoded in a function

$$F : \Sigma \rightarrow \mathbb{N}^n$$

where the $i$th component of $F(a)$ is the number of arrows going from $a$ into place number $i$, and $n$ is the total number of places. Similarly, the latter kind of information is expressed in a function

$$B : \Sigma \rightarrow \mathbb{N}^n$$

where the $i$th component of $B(a)$ is the number of arrows coming into $a$ from place number $i$.

The current state of a Petri-Net specified system may be captured completely by a vector $m \in \mathbb{N}^n$ in which the $i$th component is the number of tokens in place $i$. This vector is called the current *marking* of the Petri Net, and we will use the terms *marking* and *state* interchangeably. The initial marking (*state*) is usually denoted $M_0$.

The actions enabled at a state $M$, $En(M)$, are precisely the set of actions $a \in \Sigma$ such that the vector $M - B(a)$ has no negative components. The marking reached by firing $a$ at $M$ is defined to be the marking $M' = M - B(a) + F(a)$; this gives us our transition relation.

As an example (taken from [Nitsche and Wolper (1997)]), consider the following system, which consists of a server that behaves as follows: when the server receives a *request*, it may respond with a *result* or a *rejec*tion, depending on the availability of a resource (*free* or *lock*). If the resource is *free* when needed, the server will send a *result*; and until the *result* is sent the resource is seized by the server. If the resource is *locked*, the server responds with a *reject*ion.

This verbal description may be modelled using the Petri-Net shown in Figure 2.1.

Note that the only actions enabled at $M_0 = (0, 1, 1, 0, 0, 0)$ are *request* [because $M_0 - B(\mathit{request}) = (0, 1, 0, 0, 0, 0)$ contains no negative components] and *lock* [because $M_0 - B(\mathit{lock}) = (0, 0, 1, 0, 0, 0)$ contains no negative components]. The automaton representing the behaviour of the system is shown in Figure 2.2.

of the system.

[3] Again, this notation is peculiar to this document and is not generally used in the literature.

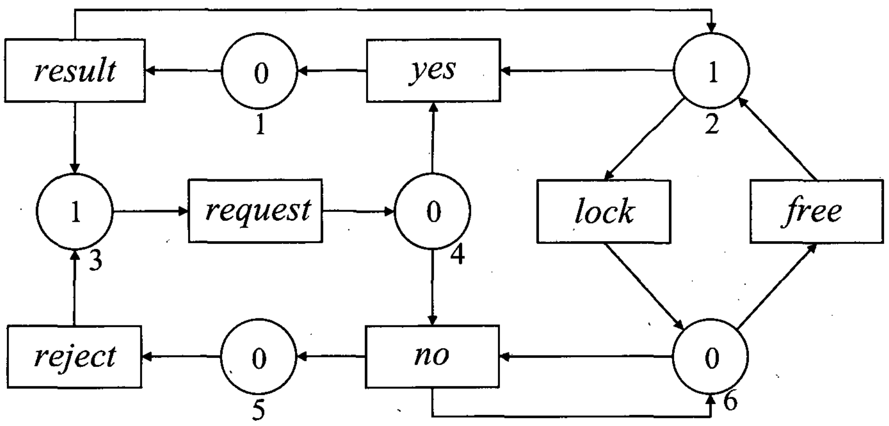


FIGURE 2.1: An Example low-level Petri Net

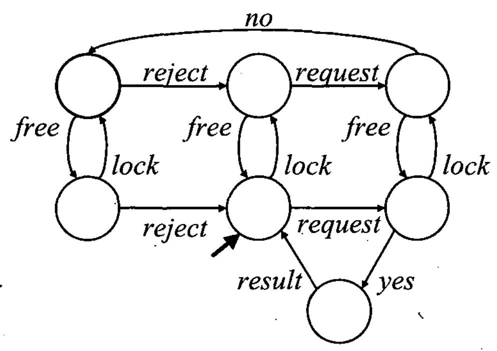


FIGURE 2.2: The behaviour of the system in Figure 2.1, as constructed by the algorithm of Algorithm 2.1

### 2.3.2 Properties

For the purposes of the results in this thesis, a property $\mathcal{P}$ is a subset of $\Sigma^{\omega}$[4][Alpern and Schneider (1985)]. Intuitively, it is a condition that we would like to check whether the behaviour of the system satisfies e.g. that the system never reaches a dead-locked state; any message sent through the system eventually reaches its intended recipient; etc.

We will describe here a formalism for describing such properties known as *Propositional Linear-time Temporal Logic* (PLTL) [Pnueli (1977)]. We present a 'cut down' version of the formalism; the most general version involves an additional set of atomic propositions and a labelling function that maps actions onto a subset of this set, all of which is superfluous to our needs. Note that the restriction of properties to those with an infinite sequence as their model unfortunately precludes the usage of properties expressed in CTL or CTL*, which has a tree-like structure as its model[Thomas (1989)].

Formally, a PLTL formula is defined recursively as follows:

*each element of $\Sigma$ is a PLTL formula, as are the atomic propositions* true *and* false;

[4] Remember, $\Sigma$ is assumed to contain the "dummy action", "#".

*if $\xi$ and $\varsigma$ are PLTL formulae, then so are*

- $(\xi) \vee (\varsigma)$, $(\xi) \wedge (\varsigma)$,
- $\neg(\xi)$, $(\xi)\, U\, (\varsigma)$*and*
- $X(\xi)$.

*Only formulae constructed in accordance with these rules are PLTL formulae.*

[the symbol $\neg$ denotes *negation*].

In addition, for notational convenience, we also introduce the abbreviations

$F(\xi) = (true)U(\xi)$ ("*eventually* $\xi$") and $G(\xi) = \neg F(\neg\xi)$ ("*always* $\xi$").

Let $x = x_1x_2... \in \Sigma^\omega$, and let $\eta$ be a PLTL formula. Then the conditions under which $x$ *satisfies* $\eta$ [written $x \models \eta$ ] are as shown below:

- If $\eta = true$ then $x \models \eta$.
- If $\eta \in \Sigma$, then $x \models \eta$ if and only if $\eta = x_1$.
- If $\eta = \neg(\xi)$, then $x \models \eta$ if and only if it is not the case that $x \models \xi$.
- If $\eta = (\xi) \wedge (\varsigma)$, then $x \models \eta$ if and only if $x \models \xi$ *and* $x \models \varsigma$.
- If $\eta = (\xi) \vee (\varsigma)$, then $x \models \eta$ if and only if $x \models \xi$ *or* $x \models \varsigma$.
- If $\eta = \chi(\xi)$, then $x \models \eta$ if and only if $x_2x_3... \models \xi$
- If $\eta = (\xi)\, U\, (\varsigma)$, then $x \models \eta$ if and only if there exists $i \in \{1, 2, ...\}$ such that $x_ix_{i+1}x_{i+2}... \models \zeta$ and for all $j < i$, $x_jx_{j+1}... \models \xi$.

Thus $(\xi)$ should be read as '$\xi$ holds immediately after the next action' and $(\xi)\, U\, (\varsigma)$ as 'eventually, $\zeta$ will hold and until then, $\xi$ must hold'. $F(\xi)$ is then simply 'eventually, $\xi$ will hold' and $G(\xi)$ is '$\xi$ must always hold'. Thus $G(F(\xi))$ is interpreted as 'it is always the case that $\xi$ will eventually hold' i.e. '$\xi$ holds infinitely often'.

The property $\mathcal{P}(\eta) \subseteq \Sigma^\omega$ specified by $\eta$ is simply the set of infinite words that satisfy $\eta$; i.e.

$$\mathcal{P}(\eta) = \{x \in \Sigma^\omega | x \models \eta\}$$

### 2.3.3 Satisfaction of Properties, and "Fairness"

There are several definitions (of varying degrees of 'fairness') of what it means for the *behaviour* of a system to *satisfy* a property $\mathcal{P}$. The most obvious would be to say that the behaviour ($\lim(L)$, from Definition 2.14) of a system satisfies $\mathcal{P}$ if and only if $\lim(L) \subseteq \mathcal{P}$; that is, all computations of the system must satisfy $\mathcal{P}$. This is called *linear-time satisfaction* [Alpern and Schneider (1985)]. Equivalently, we say that the behaviour *satisfies the property $\mathcal{P}$ linearly.*

This can, however, be slightly too restrictive: there may be computations of the system which violate the property but which are considered very unusual or unreasonable, and which we are happy to overlook. For example, the property $G(F(result))$ is violated by the system described earlier (Figure 2.2), specifically by those computations in which the resource is not *free*d sufficiently often, or often *free*d but not at the right time [the computation $lock.request.(free.lock)^{\omega}$ is an example of a computation that violates the property]. Obviously, the situation where the resource is free infinitely often but never returns a result is hardly a reasonable one and we would not accept this circumstance as a real violation of the property; the requirements for a system behaviour to satisfy the property are not "fair".Generally, fairness assumptions about the behaviour of a system are necessary for the verification of liveness properties [Owicki and Lamport (1982)] to ensure that a non-deterministic system progresses in a realistic manner, so that e.g. a process scheduler never arbitrarily starves a process forever, or, in the classic example of the Dining Philosophers problem [Lehmann and Rabin (1981)], a philosopher eventually stops eating and returns his fork.

A number of different definitions of what it means for a system to satisfy a property have been proposed, including *weak* fairness and *strong fairness* [Francez (1986)]. A computation of a system is said to be *weakly fair* if for each action $a$ that is eventually enabled throughout the computation, it holds that the action $a$ occurs infinitely often in the computation. A computation of a system is said to be *strongly fair* if for each action $a$ that is enabled infinitely often during the computation, it holds that $a$ occurs infinitely often in the computation. [Note that all strongly fair computations, then, are automatically weakly fair]. A system behaviour $B$ is then said to satisfy the property $\mathcal{P}$ under weak fairness if and only if all *weakly fair computations* of the system are in $\mathcal{P}$, with a similar definition for satisfaction under strong fairness. Appendix E provides more formal definitions of these notions.

The definition of satisfaction we will concentrate on in this thesis, however, is a relation called "satisfaction within fairness" [Nitsche and Ochsenschläger (1996); Nitsche and Wolper (1997)] [5], described in the next sub-section.

[5] In previous papers, this was referred to as "satisfaction up to liveness", or "relative liveness".

### 2.3.4 Satisfaction Within Fairness

**Definition 2.15.** We say that the behaviour of the system *satisfies* $\mathcal{P}$ *within fairness*, or equivalently, that $\mathcal{P}$ is a *relative liveness* property of the system [in either case, written "$\lim(L) \models_{WF} \mathcal{P}$"] if and only if:

$$\forall w \in pre(\lim(L)), \exists x \in cont(w, \lim(L)) : wx \in \mathcal{P}$$

In words, this states that the behaviour satisfies $\mathcal{P}$ within fairness if and only if every partial computation that may be extended to an infinite one may be extended to an infinite one satisfying the property[6]; that is, no matter which state in the system we are currently at, there is some way of satisfying the property from here. Our server satisfies the property $G(F(result))$ within fairness: it is easily checked from Figure 2.2 that no matter which state we are in, we can always reach a state where *result* is enabled and can be taken infinitely often.

Appendix E gives a more complete discussion of all of these types of satisfaction relations as well as a comparison with this new notion of satisfaction within fairness. In brief, it is shown that the four mentioned definitions of satisfactions may be arranged into a hierarchy of fairness, with linear satisfaction (the most stringent) occurring at the top, and satisfaction within fairness (the most lenient, or "fair") at the bottom:

$B$ satisfies $\mathcal{P}$ linearly $\Rightarrow$ $B$ satisfies $\mathcal{P}$ under weak fairness $\Rightarrow$ $B$ satisfies $\mathcal{P}$ under strong fairness $\Rightarrow$ $B$ satisfies $\mathcal{P}$ within fairness.

For details on deciding whether a given system satisfies a given property within fairness, see Appendix D. Very briefly, we are required at least to construct the product automaton of the automata $\mathcal{A}_L$ and $\mathcal{A}_\mathcal{P}$.

## 2.4 State-Space Explosion

And herein lies the problem: for industrial-sized specifications, the ones we are really interested in, the automaton $\mathcal{A}_L$ can be huge; given that the size of $\mathcal{A}_\mathcal{P}$ can also be exponential in the length of the PLTL formula $\eta$, and that the size of the product automaton can be as great as the product of the sizes of $\mathcal{A}_\mathcal{P}$ and $\mathcal{A}_L$, we see that the size of $\mathcal{A}_{\mathcal{P}\times L}$ can be very large; far too vast to fit within a computer's memory. This is the *state-space explosion problem*, and attempts to ameliorate this problem form the main focus of our research.

[6] In the special (and common) case where $L$ contains no maximal words, we have that $pre(\lim(L)) = L$ [as $L$ is assumed to be prefix-closed]; then the definition may be expressed as "...the behaviour satisfies $\mathcal{P}$ within fairness if and only if every partial computation may be extended to an infinite one satisfying the property..."

### 2.4.1 Current Techniques

There are already some methods to help combat this problem:

#### 2.4.1.1 Abstraction

Abstraction [Bruns (1993); Clarke et al. (1992); Graf and Loiseaux (1993); Kurshan (1994); Nitsche and Wolper (1997)] aims to hide details of the system not pertinent to the property to be verified, hopefully resulting in a simpler *abstract* system description with a correspondingly smaller state-space that can still be used to verify whether the original, *concrete* system satisfied the property.

*Data Abstraction*[Clarke et al. (1992); D.E. Long (1993); S. Graf and C. Loiseaux (1993); Graf (1994)] is one means of accomplishing this reduction. In this approach, a mapping from each of the domains of the variables comprising the state of a system to a smaller domain is created, and, consequently, the full set of *concrete* states may be mapped to a smaller set of *abstract* states. Let $S$ be the set of concrete states, $A$ the mapping that turns a concrete into an abstract state, and $\overline{S}$ the set of abstract states resulting from applying $A$ to $S$. For an abstract state $\overline{s} \in \overline{S}$, let $A^{-1}$ be the set of concrete states whose image under $A$ is $\overline{s}$. The state-space exploration is carried out at the level of abstract states: the starting state(s) in the abstract automaton are those $\overline{s_0}$ for which $A^{-1}(\overline{s_0})$ contains initial states of the concrete automaton. The transition $\overline{s} \xrightarrow{a} \overline{s}'$ is added to the abstract state space if and only there is a transition $s \xrightarrow{a} s'$ in the concrete state space with $A(s) = \overline{s}$ and $A(\overline{s}) = \overline{s}'$. Since the size of the abstract state space is clearly bounded by the size of $\overline{S}$ and this is often much smaller than $S$, the exploration generally terminates rapidly.

A classic example of this approach is that of the Dining Mathematicians, as presented in [Dams et al. (1997)]. In this example, two mathematicians, A and B, say, who alternate between *eat*ing in the communal dining area and *think*ing, have a mutual dislike of each other and have arranged a method by which they will never end up *eat*ing in the dining area at the same time by means of a flag, in the form of a positive integer $n$ to which they both have read and write access. The initial value of $n$ is arbitrary, implying multiple possible initial states and both mathematicians start off in their *think* states. Mathematician A may only proceed to the dining room if $n$ is odd; upon finishing his meal and leaving the dining area, he must set the value of $n$ to $3n+1$. Mathematician B may only *eat* if the value of $n$ is even; upon finishing, he must set the value of $n$ to $n/2$. The properties proposed for verification in [Dams et al. (1997)] are as follows:

i) There is no state in which both mathematicians are *eat*ing at once (mutual exclusion);

ii) After any state in which Mathematician A has finished *eat*ing, all subsequent execution paths will contain a state in which Mathematician B is *eat*ing (absence of individual starvation); and

iii) After any state in which Mathematician B has finished *eat*ing, all subsequent execution paths will contain a state in which Mathematician A is *eat*ing (absence of individual starvation, roles reversed).

A state must encompass the value of the flag $n$ and the individual states of the two mathematicians, each of which is in $\{eat, think\}$; thus, any state $s$ in the system will be in the set $S = \{eat, think\} \times \{eat, think\} \times \mathbb{N} - 0$. The state space is infinite, and so cannot be built exhaustively. We must use an *abstraction* coupled with *abstract interpretation*[Cousot and Cousot (1992)] in order to decide the properties.

Since parity of $n$ is so important in the system, it makes sense to take as our abstraction an abstraction that reduces the domain of $n$ to just the set $\{e, o\}$, where $n$ is mapped to $e$ if $n$ is even, and $o$ if it is odd; that is, given a concrete state $s = (a, b, n)$ where $a, b \in \{eat, think\}$ represent the states of the two mathematicans, respectively, at $s$:

$$\begin{aligned} A(s) &= (a, b, e) \quad \text{if } n \text{ is even} \\ &= (a, b, o) \quad \text{if } n \text{ is odd} \end{aligned}$$

Thus, any abstract state must be contained in $\{eat, think\} \times \{eat, think\} \times \{e, o\}$. The exploration of the state space is now quite simple: The initial value of $n$ is arbitrary and A and B both start off *think*ing, so the initial abstract states are $(think, think, e)$ and $(think, think, o)$, as $n$ is abstracted to being either just even or odd. At $(think, think, o)$, A is allowed to *eat* as this corresponds to $n$ being odd in the concrete system, and *eat*ing leads to $(eat, think, o)$. When A finishes *eat*ing, he sets $n$ to $3n + 1$ which is always even when $n$ is odd, as is the case here, and so leads to state $(think, think, e)$. Since this corresponds to concrete states with even $n$, B may now *eat*, leading us to the abstract state $(think, eat, e)$. Now, when B finishes, he sets $n$ to $n/2$, and this value may be odd *or* even, and so when B finished *eat*ing we may be in the state $(think, think, o)$ or $(think, think, e)$, so we must include transitions to both these states. The construction continues according to these rules until no new abstract states are created, rapidly resulting in the state space of Figure 2.3 (adapted from [Dams et al. (1997)]).

It was shown in [Dams et al. (1997)] that this abstraction and the resulting state space in Figure 2.3 are sufficient to prove i) (there are no states in Figure 2.3 where A and B are *eat*ing at once) and ii) (the only continuation from the only state where A is *eat*ing, $(eat, think, o)$, leads to the state $(think, eat, e)$ after two transitions). It is not able to prove property iii), however: according to Figure 2.3, we may cycle forever between the states $(think, think, e)$ and $(think, eat, e)$, never allowing A to *eat*. In general, it holds that any computation that can occur in the concrete system can also occur in the

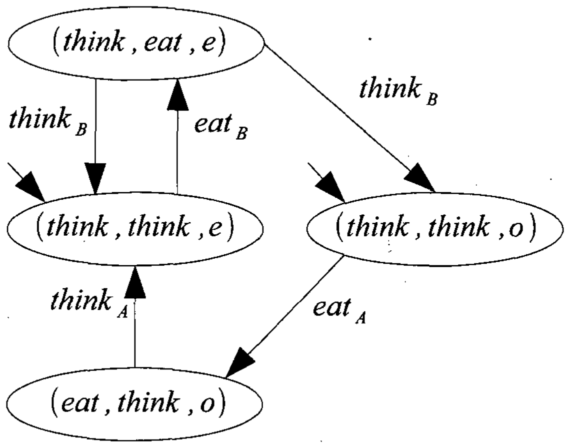


FIGURE 2.3: Abstract state-space of the Dining Mathematicians problem

abstract version, but that some computations may occur in the abstract system that can not occur in the concrete.

The abstraction technique we will focus on is *behaviour abstraction*[Nitsche (1998)]. Like data abstraction, this approach also aims to reduce the state-space but, whereas data abstraction achieves this by reducing the domains of the variables comprising a state, directly reducing the number of possible abstract states, behaviour abstraction reduces the state-space indirectly by reducing the domain of the *actions* the system can perform, giving a simpler abstract *behaviour* that can hopefully be represented by a smaller automaton. For example, consider an extremely detailed specification of a communications network, including low-level specifications of the protocols used to send messages, descriptions of handshaking between terminals etc. in addition to the high-level specifications (rules for dealing with clients, when to send messages etc.): the behaviour is very complex, and the automaton representing this behaviour will likely be very large. However, the correctness criterion is whether or not the network 'serves' the client correctly i.e. it is a condition on the high-level behaviour only (the behaviour that is visible to the user) and so the low-level details are relegated to mere 'implementation details' that need not be considered in the verification step. Thus, we can ignore the low-level (invisible to the user) actions and concentrate on the actions that are directly related to the interaction with the client.

More formally, the group of low-level actions contributing to an elementary high-level operation (for example, sending a message) can be coalesced into one high-level action (called *send_message*, for example) or perhaps hidden if they are entirely unimportant to the high-level behaviours being verified. In short, behaviour abstraction ignores certain actions in the implementation and focuses only on actions that are visible to the outside world.

Abstractions of this kind are carried out by using an *abstraction* (or *abstracting*) *homo-*

*morphism*[Nitsche (1998)].

**Definition 2.16.** Let $\Sigma$, $\Sigma'$ be two alphabets. Define $\Sigma^\infty = \Sigma^\omega \cup \Sigma^*$ (the set of all finite and $\omega$-words consisting of letters from $\Sigma$), and similarly for $\Sigma'^\infty$. Then an *abstracting homomorphism*, $h$, is a mapping

$$h : \Sigma^\infty \to \Sigma'^\infty \cup \{\varepsilon\}$$

such that the following conditions all hold:

i) If we restrict $h$ to $\Sigma$, we obtain a total function $h : \Sigma \to \Sigma' \cup \{\varepsilon\}$

ii) $\forall x, y \in \Sigma^*$, $z \in \Sigma^\omega$ we have that

$$h(x)h(y) = h(xy)$$

and

$$h(x)h(z) = h(xz)$$

iii) $h(x)$ is undefined for any $x \in \Sigma^\omega$ such that $h(x)$ would otherwise be finite (an abstracting homomorphism cannot map an infinite word to a finite one).

The set $\Sigma'$ is called the set of *abstract actions* and corresponds to the high-level operations that the abstracted system may perform when we have ignored the low-level ones or combined them into a single new abstract action ; in contrast, we call the original set $\Sigma$ the set of *concrete actions*[7]. In our examples, $\Sigma'$ will generally be a subset of $\Sigma$, but our analysis does not require this to be the case so we allow $\Sigma'$ to be an arbitrary finite alphabet. Condition i) asserts that $h$ maps concrete actions to abstract actions. Condition ii) asserts that the concatenation of the abstractions of two finite words (or a finite word and an $\omega$-word) is equal to the abstraction of the concatenations. Condition iii) is more complex, and will be dealt with separately. The original behaviour, $\lim(L)$, is called the *concrete behaviour*, and the new behaviour $\lim(h(L))$ the *abstract behaviour*. Coalescing several concrete actions into one (non-hidden; i.e. not equal to $\varepsilon$) abstract action is called *action renaming*; mapping actions to the hidden word $\varepsilon$ is called *action hiding*.

The formulation given in Definition 2.16 was, for the purposes of [Nitsche and Ochsenschläger (1996)] (which was focussed on proving properties of such mappings on $\omega$-words) the most natural, but for our purposes (which are more focussed on mappings acting

[7] To preserve the compatibility with the upcoming Theorem 2.18, we define $h(\#) = \#$, and assume (as we did with $\Sigma$) that $\Sigma'$ contains the dummy action, "#".

on *individual actions*, rather than $\omega$-words) we will use a different, equivalent formulation. We note that a function $h : \Sigma \to \Sigma' \cup \{\varepsilon\}$ defined on individual actions (which satisfies Definition 2.16i)) can be extended uniquely to an abstracting homomorphism as defined in Definition 2.16 by the inductive application of Definition 2.16ii) to build up a mapping from $\Sigma^\infty$ to $\Sigma'^\infty$, followed by the culling of any created mappings $h(x)$ where $x \in \Sigma^\omega$ and $h(x)$ is finite as required by Definition 2.16iii) and so, we will, throughout this thesis, define abstracting homomorphisms simply as functions from $\Sigma$ to $\Sigma' \cup \{\varepsilon\}$.

The condition Definition 2.16iii) is in need of some justification; briefly, it was required for the proofs in [Nitsche (1994)] and it can be shown that, within the context of satisfaction within fairness under abstraction, the exclusion of words of this form is not problematic: a word $x \in \Sigma^\omega$ that is mapped to a finite word by $h$ represents an infinite sequence of actions that, when abstracted, become invisible after a certain finite point under the abstraction. In our analysis, such computations are represented instead by a computation $y$ that shares a prefix with $x$ but which always continues to be visible under the abstraction - more specifically, $y$ is such that $h(y)$ is infinite and $h(x) \prec h(y)$. Should a system behaviour contain such an $x$ without a corresponding $y$, then the abstracting homomorphism will fail a condition called *weak continuation-closure* (defined later) and we will be unable to verify the satisfaction within fairness of any property under the given abstracting homomorphism. Suffice to say that in practice, the condition of Definition 2.16iii) offers no additional restriction on the possible choices of the function $h : \Sigma \to \Sigma' \cup \{\varepsilon\}$.

Informally, an abstracting homomorphism is 'good' if there is plenty of action renaming and hiding (hiding being the more satisfactory of the two) i.e. if lots of concrete actions are mapped into the same abstract action, and 'bad' if this does not occur that much; the two extremes occur on the one hand when all actions are hidden (and so all information is lost) and on the other when $h$ is the identity (in which case we have not simplified the system at all, and have gained no reduction in the size of the state space).

Construction of the automaton $\mathcal{A}_{h(L)}$ representing the abstract behaviour is simple: in the for each transition in the concrete automaton, $\mathcal{A}_L$, replace the label, $a \in \Sigma$, say, with its abstract image, $h(a) \in \Sigma'$ and determinise the result. Determinisation of this automaton serves the dual purpose of collapsing the state-space down to a hopefully much smaller size, and of making the automaton amenable to decidability tests for weak-continuation closure and satisfaction within fairness of a property, both of which require a determinised automaton to work on; see Appendices A and D, respectively, for more details on these algorithms.

Formally, the process of determinising $\mathcal{A}_L$ after the re-labelling is $O(2^{|\mathcal{A}_L|})$ and can lead to exponential blow-up, but for most practical cases it is far, far less than this very pessimistic upper bound, and so the computation of $\mathcal{A}_{h(L)}$ is usually feasible and, more importantly, potentially very useful. The reason that the resulting determinised automaton is often smaller than the original is that usually the abstracting homomorphism is chosen explicitly so that the observable abstract behaviour is much simpler than the concrete behaviour, often by hiding all those actions that correspond to complex low-level work that the end user never sees. For example, if there were a system with a complex communications channel with the property that sent messages are always received and no message is sent until the previous one has been received, then the observable result, under an abstracting homomorphism that hides all actions apart from sending out a message over the channel (mapped to *send_msg*, say) and fully receiving a sent message (mapped to *recv_msg*, say), would be the prefix-closure of $(send\_msg.recv\_msg)^*$, which can be represented by an automaton with just two states. For further justification of the statement that the abstract automaton is usually smaller than the concrete, including experimental results, see e.g. [Ultes-Nitsche (1998)].

We note at this point an important difference between the techniques of data and behaviour abstraction regarding efficiency - both techniques ultimately map sets of concrete states on a single abstract state, but with data abstraction this mapping is very explicit: given a concrete state, we can immediately map it to its abstract image (for example, in the Dining Mathematicians example, the concrete state $(eat, think, 5)$ is clearly mapped to the abstract state $(eat, think, o)$ as 5 is odd), or map an abstract state to the corresponding set of concrete states. As a consequence, we may perform our state-space exploration directly at the level of abstract states, of which there are generally far less than of concrete states, resulting in a quickly terminating exploration.

In contrast, with behaviour abstraction the relation between a concrete state and its abstract image is much more opaque, depending more on the sequences of actions that lead in to and out of the state than the properties of the state itself, and so we cannot even know the set of final abstract states until the we perform the determinisation step (which accomplishes the final mapping of sets of concrete states to abstract states) which appeared to be only doable *after* the *whole* concrete state space has been constructed. That is, even though the final set of abstract states may be quite small, we still must construct the whole state space in order to find them. The main result of this thesis reduces the impact of this by allowing us to use a reduced representation of the concrete state space in place of the full state space, but it remains the case that data abstraction usually terminates more rapidly and does not need to explore more states than are finally present in the abstract automaton.

### 2.4.1.2 Abstraction with Satisfaction Within Fairness

Properties are now defined for the abstract (high-level) behaviour (i.e. $\mathcal{P} \subseteq \Sigma'^{\omega}$ ), and we would like to be able to infer whether the concrete behaviour satisfies a given property within fairness based on whether the abstract behaviour satisfies that property. If such an inference is justified, i.e. if

$$\lim(L) \models_{WF} h^{-1}(\mathcal{P}) \Leftrightarrow \lim(h(L)) \models_{WF} \mathcal{P}$$

holds, where $h^{-1}(\mathcal{P}) = \{x \in \Sigma^{\omega} | h(x) \in \mathcal{P}\}$ , then we say that the abstraction $h$ *preserves the property $\mathcal{P}$ under satisfaction within fairness*, or simply that *$h$ preserves the property.* Examples can be given in which a choice of $h$ does *not* in fact preserve the property (i.e. the abstract behaviour satisfies a property within fairness while the concrete behaviour does not satisfy the corresponding concrete-level property); see Figure 2.4.

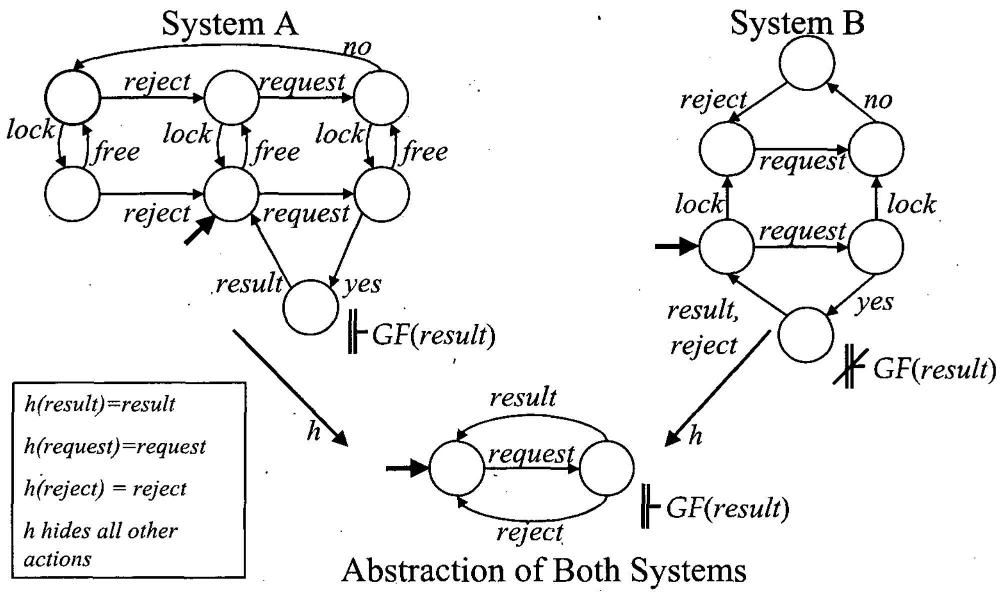


FIGURE 2.4: An example that shows that we cannot always deduce properties of the concrete system based on its abstraction.

It was shown in [Nitsche and Ochsenschläger (1996)] that whether or not $h$ will preserve satisfaction within fairness can be decided simply by testing whether $h$ satisfies a condition called *weak continuation-closure.*

### 2.4.1.3 Weak Continuation-Closure

A *weakly-continuation closed homomorphism* $h$ is contrasted with a *continuation closed* $h$, defined as follows: $h$ is *continuation closed* on $L$ if and only if it satisfies the (fairly restrictive) condition

$$\forall w \in L, h(cont(w, L)) = cont(h(w), h(L))$$

It was shown in [Nitsche and Ochsenschläger (1996)] that this is a needlessly strong condition for preserving properties and weak continuation closure was defined as follows:

**Definition 2.17.** $h$ is WCC on $L$ if and only if:

$$\forall w \in L, \exists v \in cont(h(w), h(L)) : cont(v, h(cont(w, L))) = cont(v, cont(h(w), h(L)))$$

This is a rather difficult concept to grasp; the intuition is that $\forall w \in L$, we do not have immediate continuation closure, but by following some string of abstract actions from $h(w)$ we 'eventually' have continuation closure at some later state. All we need to know about WCC homomorphisms, though, is whether a given homomorphism is WCC on a language is decidable[Ochsenschläger (1994), Appendix A], with some useful, efficiently checkable sufficient conditions (for example, it can be shown that if the automaton representing $L$ is strongly-connected, then *any* abstracting homomorphism $h$ will be WCC on $L$) and that they are, as mentioned, precisely the class of homomorphisms that preserve satisfaction within fairness; that is:

**Theorem 2.18.** *Let $L \in \Sigma^*$ be a prefix-closed regular language containing no maximal words, $h : \Sigma \to \Sigma' \cup \{\varepsilon\}$ an abstracting homomorphism defined on $L$, and $\mathcal{P} \in \Sigma'^{\omega}$ a property. Then*

$$\lim(L) \models_{WF} h^{-1}(\mathcal{P}) \Leftrightarrow \lim(h(L)) \models_{WF} \mathcal{P}$$

*if and only if $h$ is weakly-continuation closed (WCC) on $L$.*[8]

Note that we will assume throughout this thesis that $L$ is a regular language i.e. that $L$ may be represented by a finite automaton, unless otherwise stated. Whether or not the result could be extended to include non-regular $L$ was an open question, finally answered decisively in Appendix C.

From Theorem 2.18, and the fact that the behaviour of System B from Figure 2.4 does not satisfy a property within fairness whereas its abstraction does, we can deduce that $h$ was in fact *not* WCC on the language $L$ representing the behaviour of B, and it may be instructive to digress briefly and examine why.

Let $L$ be the language representing the behaviour of System B, and $\mathcal{A}$ the automaton representing this language (so $\mathcal{A}$ is as shown in Figure 2.5 a)). Both [Ochsenschläger (1994)] and Appendix A hint that the weak-continuation closure of a homomorphism $h$ on a language $L$ represented by an automaton $\mathcal{A}$ depends on strongly-connected bottom components of $\mathcal{A}$. We will pick a state $s$ in the single strongly-connected bottom component and examine the abstraction of its continuation within $L$ i.e. examine $h(cont(s, L))$.

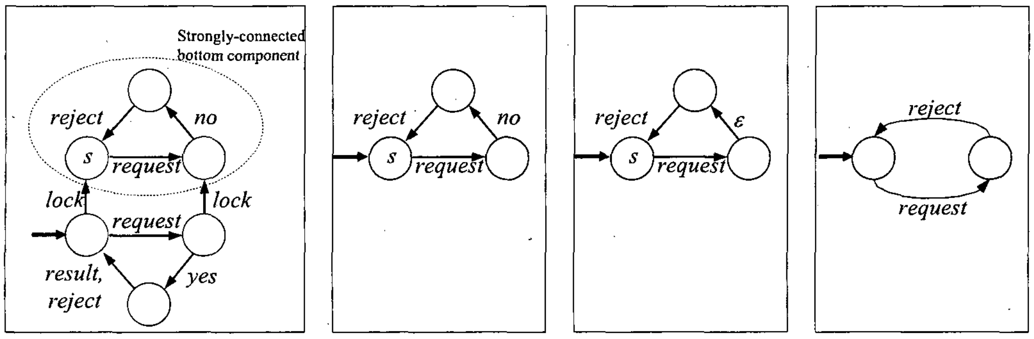
The first step is to compute $cont(s, L)$: from the comment immediately after Definition 2.10, we can do this by re-setting the initial state of $\mathcal{A}$ to $s$ and then removing all states and transitions that are no longer reachable from this new initial state $s$, as

[8] Technically, this result holds only when $L$ has no maximal words ($\mathcal{A}_L$ has no deadlock states); however, as we have seen, the inclusion of the dummy action "#" circumvents this restriction.

they no longer contribute anything. The result of this step is shown in Figure 2.5 b). Next, we replace all the transitions labels $a \in \Sigma$ with their abstract images, $h(a) \in \Sigma'$; this yields the automaton representing $h(cont(s, L))$, shown in Figure 2.5 c). Finally, we determinise this automaton, to obtain the more compact and manageable representation of $h(cont(s, L))$, as shown in Figure 2.5 d). Note that the action *result* never occurs in the language $h(cont(s, L))$. Next, note that the word $w = lock$ leads from $s_0$ to $s$ in the original automaton, $\mathcal{A}$; from Note 2.1, we have that $cont(s, L) = cont(w, L)$, from which it follows that $h(cont(s, L)) = h(cont(w, L))$. Since $h(w) = \epsilon$, we have

$$cont(h(w), h(L)) = cont(\epsilon, h(L)) = h(L)$$

So according to Definition 2.17, in order for $h$ to be WCC on $L$, we would need there to be a word $v \in h(L)$ such that $cont(v, h(cont(s, L))) = cont(v, h(L))$ (the automaton representing $h(L)$ is shown at the bottom of Figure 2.4). However, there is no such $v$; as noted previously, the abstract action *result* never occurs in $h(cont(s, L))$, but it always appears in at least one word in $cont(v, h(L))$ for any $v \in h(L)$, as from any state in the automaton representing $h(L)$ we can always reach a state where *result* is enabled and taken. Thus, there is a word $w$ in $L$ which, when followed, restricts the subsequent behaviour so much that it cannot ever coincide with the abstract behaviour $h(L)$ (in this case, because we can no longer perform an action that can always be performed by the full abstract behaviour), and this is why $h$ is not WCC on $L$ .



(a) Original behaviour (b) $cont(s, L)$ (c) $h(cont(s, L))$ (d) $h(cont(s, L))$ (determinised)

FIGURE 2.5: Abstraction of a Strongly-Connected Bottom Component

So if $h$ is WCC on $L$, then we can preserve satisfaction within fairness of properties *sufficiently* under abstraction, and thus may deduce properties of the concrete behaviour based on the abstract behaviour. The size of the final product automaton, $\mathcal{A}_{\mathcal{P} \times h(L)}$, is usually significantly smaller than that for the concrete behaviour $\lim(L)$ and property $h^{-1}(\mathcal{P})$. The problem is, however, that we must currently construct the entire (prohibitively large) concrete automaton $\mathcal{A}_L$ in order to construct $\mathcal{A}_{h(L)}$, which may not be feasible . Thus, in the unusual case where the state-space of the system is effectively constructible, we do gain some advantage from this approach; constructing the product automaton for deciding satisfaction within fairness can be much easier for the abstract behaviour than for the concrete. However, in the more likely case where the state-space is simply too big, we gain nothing. We will return to this problem later on, in Chapter 3.

#### 2.4.1.4 Partial-Order Methods.

Consider a state in a system in which a number of actions $a_1, a_2, ..., a_n$ are enabled, and imagine further that it does not matter which order a pair of these actions is executed in: we end up in the same state in the automaton either way. This can happen if, for example, neither action interferes with any resources used by the other action. Then a straightforward application of the algorithm for constructing the state-space of the system (Algorithm 2.1) will implicitly consider all orderings, or *interleavings* of these *independent* actions [there are up to $n!$ such interleavings in the worst case in which all $n$ actions are all mutually independent], when we do not really gain much from doing so as the end result will only be considered once . The *persistent set selective search* of Godefroid and Wolper [Godefroid (1991, 1995); Godefroid and Wolper (1993); Wolper and Godefroid (1991)], a particular partial-order method, gives us a method of avoiding consideration of all these interleavings of independent actions, giving a reduced automaton that preserves a restricted class of properties.

We begin now to formalize these concepts.

**Definition 2.19.** Actions $a, b \in \Sigma$ are *independent* [Godefroid (1991, 1995); Godefroid and Wolper (1993); Wolper and Godefroid (1991)] at $w \in L$ if they satisfy the following pair of properties:

i) $cont(wab, L) = cont(wba, L)$

ii) if $b \in cont(w, L)$ then $a \in cont(w, L) \Leftrightarrow a \in cont(wb, L)$, and if $a \in cont(w, L)$ then $b \in cont(w, L) \Leftrightarrow b \in cont(wa, L)$.

If a pair of actions $a, b \in \Sigma$ are independent at all $w \in L$, then we simply say that $a$ and $b$ are *independent.*

The first condition asserts that the order in which we carry out the actions is unimportant; they both lead to the same state (the same *continuation*). The second states that the two actions may neither enable nor disable one another.

As always, these definitions in terms of words $w \in L$ are extended to states $s$ in the automaton representing $L$, as per Note 2.1.

**Definition 2.20.** An *independence relation* $\Delta$ over a language $L \subseteq \Sigma^*$ is a relation $\Delta \subseteq \Sigma \times \Sigma \times L$ such that $(a, b, w) \in \Delta$ implies that $a$ and $b$ are independent at $w$ in $L$.

As an aside, note that independence relations are always symmetric (that is, $(a, b, w) \in \Delta \Leftrightarrow (b, a, w)$ for all $a, b \in \Sigma$ and $w \in L$) by definition, but are not necessarily reflexive nor transitive - an example of an independence relation that is neither reflexive nor transitive is given in Section 3.4.

For a given $w \in L$ and independence relation $\Delta$ over $L$, we impose an equivalence relation $\equiv_{\Delta,w}$ on words in $cont(w, L)$ such that the words $w', w'' \in cont(w, L)$ satisfies $w' \equiv_w w''$ if and only if $w'$ may be transformed into $w''$ by a sequence of transpositions of adjacent independent actions. Two such equivalent words are said to be *trace equivalent* at $w \in L$. More formally, we have:

**Definition 2.21.** Let $w \in L$. Then we define a relation $\equiv_{\Delta,w} \subseteq cont(w, L) \times cont(w, L)$ by saying that $w' \equiv_{\Delta,w} w''$ if and only if there exists a sequence of words $w_1, w_2, ..., w_n$ for some $n \geq 1$ with $w' = w_1$ and $w'' = w_n$ and such that, for each $1 \leq m < n$ we may write

$$w_m = x a_m b_m y$$

and

$$w_{m+1} = x b_m a_m y$$

where $(a_m, b_m, wx) \in \Delta$.

The set of words that are trace equivalent at $w \in L$ is called the *trace* at $w$:

**Definition 2.22.** Given $w \in L$ and an independence relation $\Delta$ over $L$, the set of words in $cont(w, L)$ which are trace equivalent to $w' \in cont(w, L)$ at $w$ is called the *trace containing $w'$ at $w$*, and is denoted $[w']_{\Delta,w}$.

Partial-order methods are geared towards discarding some (but not all) of the words in the same trace during construction of the automaton, giving a reduced automaton representing a language that we will call $R_L^{\Delta}$. The algorithm for constructing such a reduced automaton is called the *persistent-set selective search* [Godefroid and Wolper (1993)] and may be expressed as in Algorithm 2.2

The function Persistent_set($s$) returns a set of actions that is a *persistent set* at $s$. A *persistent set* at a state $s$ (or equivalently, at a word $w \in L$: see Note 2.1) is a non-empty subset of the enabled actions at $s$ such that *all actions in the persistent set $P$ at $s$ are independent of the actions that could possibly become enabled at any state that could be reached from $s$ without firing any actions in $P$*. More formally (and hopefully, more clearly!)

**Definition 2.23.** Let $L \subseteq \Sigma^*$ be a language over an alphabet $\Sigma$, $\Delta \subseteq \Sigma \times \Sigma \times L$ be an independence relation over $L$. Then $P \subseteq En(w)$ is *persistent at* $w \in L$ if it is non-empty and if the following condition holds:

```
1:  $Q \leftarrow \{s_0\}$, $stack \leftarrow \phi$;
2:  push ($s_0$) onto $stack$;
3:  sub DFS()
4:      $s = top(stack)$;
5:      $Q \leftarrow Q \cup \{s\}$;
6:      let $T$ =Persistent_Set($s$);
7:      if $T = \phi$ then          // The state $s$ must be a deadlock
8:          $\delta \leftarrow \delta \cup (s, \#, s)$;  // state. Add a loop to itself marked
9:      endif                       // with the dummy label, "#"
10:     for each action $a$ in $T$ {
11:         find state $s'$ such that firing $a$ at $s$ leads to $s'$;
12:         $\delta \leftarrow \delta \cup (s, a, s')$;
13:         if $s' \notin Q$
14:             push ($s'$) onto $stack$;
15:             call DFS();
16:         endif
17:     }
18:     pop $s$ from $stack$;
19: end sub
```

**Algorithm 2.2** – The Persistent-set Selective Search

Let $p$ be any action in $P$, $v = a_1a_2...a_n$ be any word in $(\Sigma - P)^* \cap cont(w, L)$. Then we require, for each $m = 1, 2, ..., n$ that $(p, a_m, wa_1a_2..a_{m-1}) \in \Delta$. As with the notion of WCC, this is a rather difficult definition to grasp, and the original *raison d'être* for persistent sets may be a useful aid. The definition was motivated by the observation that, if $P$ were persistent at $s$, then no word $v$ in $(\Sigma - P)^* \cap cont(w, L)$ could lead to a deadlock state (because following $v$ would have to disable all actions in $P$, which is impossible since all actions in $v$ are independent of all actions in $P$) and so we would not need to consider them in our search for deadlocks.

We present the alternate (but equivalent) definition in terms of states in the hopes that this will further clarify the concept for the reader:

**Definition 2.24.** Let $L \subseteq \Sigma^*$ be a language over an alphabet $\Sigma$ represented by an automaton $\mathcal{A}$ with set of states $S$, $\Delta \subseteq \Sigma \times \Sigma \times S$ be an independence relation over $L$. Then $P \subseteq En(s)$ is *persistent at the state $s$ of $\mathcal{A}$* if it is non-empty and if the following holds.

Let $p$ be any action in $P$, $v = a_1a_2...a_n$ be any word in $(\Sigma - P)^* \cap cont(s, L)$; then there is a sequence of states $s = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3...s_n \xrightarrow{a_n} s_{n+1}$ in $\mathcal{A}$, and none of the $a_i$ is in $P$. Then we require, for each $m = 1, 2, ..., n$, that $(p, a_m, s_m) \in \Delta$.

From the definiton, the full set of enabled states at $s$, $En(s)$, is always persistent, so such a function Persistent_set($s$) does exist. Persistent sets are relatively simple to construct with Petri-Net specified systems [Valmari (1991a,b), Section 4.3.3]. The full set of enabled states at $s$ is referred to as the *trivial persistent set.*

As an example of a persistent set, consider the Petri Net shown in figure Figure 2.6. We will attempt to find a persistent set at the state $s = (1, 0, 1)$. There are precisely two actions enabled at the state $s$; $a$ and $c$ (so $En(s) = \{a, c\}$). Thus, a *non-trivial* persistent set at $s$, if it exists, will be either $\{a\}$ or $\{c\}$; we will examine both candidates in turn. The independence relation is as follows: $a$ and $b$ are *dependent* as firing $a$ will always disable $b$ and vice versa (as each actions removes the single token from the place marked 1, whose presence is necessary for the other action to fire). $a$ and $c$ do not share access to any place, and so it is impossible for one to directly affect the other; thus, $a$ and $c$ are *independent.* Finally, $b$ and $c$ are *dependent,* as firing $c$ will enable $b$. So $\Delta = \{(a, c), (c, a)\}$..

So take $P = \{a\}$ as our candidate persistent set. Then, according to Definition 2.24, we must examine all words in $(\Sigma - P)^* \cap cont(s, L)$, and make sure that they do not contain any actions that $a$ is dependent on. Since the only action that $a$ is dependent on is $b$, this reduces to making sure that $(\Sigma - P)^* \cap cont(s, L)$ does not contain any words that include the action $b$.

We construct $cont(s, L)$ by exploring from $s$ using the state space exploration algorithm of Algorithm 2.1, modified so that the starting state is now $s$. The resulting automaton is shown in Figure 2.7a). To find $(\Sigma - P)^* \cap cont(s, L)$, we remove from this automaton any transitions labelled with an action from $p = \{a\}$, and then remove all states and transitions that are no longer reachable from $s$: the result of this process is shown in Figure 2.7b). Straightaway, we see that the word $cb \in (\Sigma - P)^* \cap cont(s, L)$, and since $(a, b) \notin \Delta$ i.e. $a$ and $b$ are not independent, we see that $P = \{a\}$ is *not* a persistent set at the state $s$.

Let us move on to the only other candidate: $P = \{c\}$. The only action $c$ is dependent on is $b$, so we must ensure that $b$ does not occur in any word in $(\Sigma - P)^* \cap cont(s, L)$. We compute $(\Sigma - P)^* \cap cont(s, L)$ in an identical manner to the last case; remove from $cont(s, L)$ any transitions with label with an action in $P = \{c\}$, and remove all states/ transitions that are no longer reachable from $s$. The result of this operation is shown in Figure 2.7c). This time, we see that no word in $(\Sigma - P)^* \cap cont(s, L)$ contains $b$ (in fact, this set consists of just the words $\epsilon$ and $a$), so the set $P = \{c\}$ *is* persistent at the state $s$.

It should be noted that this is a very artificial example of the construction of a persistent set: in a real-life example, we would not proceed by trying out each non-empty proper subset of $En(s)$ in turn to see if it is persistent, as this would be unfeasible with a large $En(s)$. Nor would we explicitly construct the automata representing $cont(s, L)$ or

$(\Sigma - P)^* \cap cont(s, L)$, as this would negate the whole purpose of partial order reduction, which aims to cut down the amount of state-space exploration as much it can. A much more efficient (though less "optimal", as it is not guaranteed to find the smallest persistent set at a given state) method, the *stubborn-set technique*[Valmari (1991a,b)] is detailed in Chapter 4.

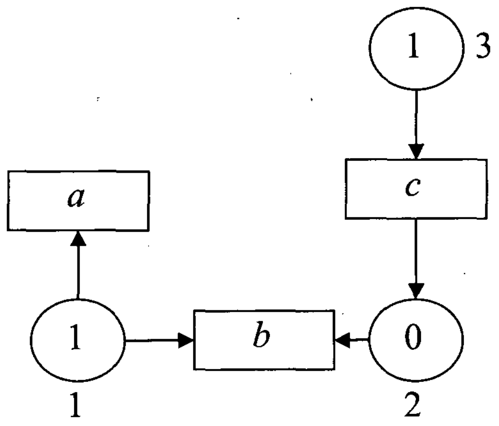
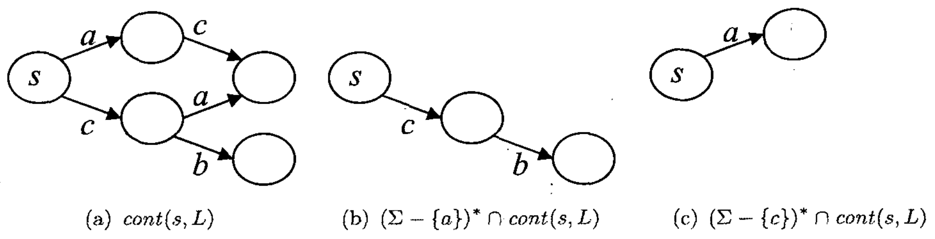


FIGURE 2.6: A Petri-net Based Persistent Set Example



s
a
c
c
a
b
s
c
b
s
a
(a) $cont(s, L)$
(b) $(\Sigma - \{a\})^* \cap cont(s, L)$
(c) $(\Sigma - \{c\})^* \cap cont(s, L)$

FIGURE 2.7: Continuations of Figure 2.6 for Candidate Persistent Sets

It can be shown that the reduced automaton resulting from a persistent set selective search will contain a deadlock if and only if the full automaton includes one [Godefroid and Wolper (1993)] or, equivalently, that the language represented by the reduced automaton will contain a maximal word if and only if $L$ contains one. However, because we are discarding a lot of information concerning the ordering and indeed, the *presence* of actions, this is about the only property that we can guarantee will be preserved during the reduction[9]. There are apparently newer methods wherein the construction of the reduced automaton is guided by the property one wishes to check in such a way as to ensure the property is preserved, but in general a blind application of the persistent-set selective search will likely destroy any properties one would wish to check.

[9]Strictly, we should say that this is the only *liveness* property that we can guarantee will be preserved as, by noting that we can modify the system specification so that it deadlocks as soon as the required safety property (say, "no division by zero", for example) is violated, we see that many safety properties can be reduced to deadlock detection [Godefroid (1995); Varpaaniemi (1993)].

## 2.5 Summary

We have described key concepts from Automata and Language Theory, and defined the *behaviour* of a system to be the *Eilenberg Limit* of a prefix-closed regular language $L$ over an alphabet $\Sigma$ of actions performable by the system. Equivalently, the behaviour is the full set of *computations* of the system, where each computation is the result of a finite (*partial*) computation of the system, extended to infinity.

We have defined the *satisfaction of a property by a behaviour* under various definitions of *fairness*, and in particular the concept of satisfaction of a property by a behaviour *within fairness*, a relation which, informally, states that the behaviour of a system satisfies a given property if and only if, after following any finite sequence of actions in the system (or equivalently, performing any partial computation of the system), we may always extend this finite sequence to an infinite one which satisfies the property. The class of properties for which this result is applicable is the class of properties which have infinite sequences as their base such as e.g. PLTL. Importantly, CTL does not fall within this class of properties.

After introducing *behaviour abstraction*, a technique of abstraction that achieves reduction in the state-space of the system by a combination of hiding individual actions in $\Sigma$ and merging groups of actions into one abstract action by means of an *abstracting homomorphism*, we observed that we can deduce which properties are satisfied within fairness by the original behaviour by examining those properties satisfied within fairness by the abstract system, as long as the abstracting homomorphism fulfils a condition called *weak continuation-closure*. Since the automaton representing the abstract behaviour could be substantially smaller than that of the original behaviour, this could give a gain in efficiency. However, this gain is almost completely negated by the fact that in order to construct the abstract behaviour, we must first construct the original behaviour, which may be unfeasible.

A candidate solution to this problem involves the incorporation of *partial-order reduction* techniques. These techniques aim to reduce the size of the state-space by avoiding having to consider all (redundant) interleavings of sequences of actions which are deemed to be *independent* of each other. This is achieved by following, at each state reached during the construction of the state-space, a subset of the full set of actions enabled at that state called a *persistent set*. Calculation of small persistent sets is in general non-trivial, but with e.g. *Petri-net* specified systems, there exist effective means for constructing reasonably small persistent sets, which we will detail in Chapter 4.

This concludes the summary of the existing techniques; we now go on to present a chapter on the main result of this thesis [St James and Ultes-Nitsche (2001)], which aims to combine partial-order methods and abstraction in order to solve the problem of having to compute the full behaviour prior to constructing the abstract behaviour.

# Chapter 3

# Combining Partial-order Reduction and Abstraction

## 3.1 Overview

As mentioned, accepting only a subset of the words in each trace will usually destroy properties of the system, *on the concrete level.* However, if we restrict our definition of an independence relation $\Delta$ so that a pair of actions are in $\Delta$ only if they satisfy the requirements of Definition 2.19 and, in addition, are 'compatible' under the abstraction [i.e. have the same abstracted image], then it is easily seen that all words in the same trace will have identical images under $h$. Thus, discarding some of these words should not destroy properties *on the abstract level,* since their images are represented by any of their fellow trace-members that we decided to accept. This insight led to the result presented in [Ultes-Nitsche (1999)].

The essence of this paper was that, having modified the independence relation so that independent actions must be $h-$compatible[1], the reduced automaton (representing the reduced language, $R_L^\Delta$, which we will describe more fully in Definition 3.4) resulting from an application of a (modified) persistent-set selective search would preserve sufficiently properties of the concrete behaviour under abstraction. Furthermore, it was shown that $h$ is WCC on $R_L^\Delta$ if and only if $h$ is WCC on $L$; thus, $R_L^\Delta$ can be used entirely in place of $L$ when verifying relative liveness properties without any detrimental effects whatsoever. More formally, the main result of the paper may be expressed as:

$$\lim(h(R_L^\Delta)) \models_{WF} \mathcal{P} \Leftrightarrow \lim(L) \models_{WF} h^{-1}(\mathcal{P})$$

*if and only if $h$ is WCC on $R_L^\Delta$.*

[1] More precisely, we first form our standard independence relation $\Delta$ (that relation containing pairs of actions that satisfy the requirements of Definition 2.19) and then make it $h$-compatible by removing from $\Delta$ any pairs $(a, b) \in \Delta$ that do not satisfy $h(a) = h(b)$.

The benefit here of course is that we no longer need to construct the automaton representing the full set of behaviours of the system prior to abstraction; the construction of the partial-order reduced automaton is sufficient. In other words, we do not have to construct the entire state-space in order to derive properties about it.

The main result of this thesis, presented at PODC'01 [St James and Ultes-Nitsche (2001)], refines this result and makes it much more practical, and is the subject of this chapter. A small and simple Petri-net based example of the usage of this result is given in Section 3.4, but it should be stressed that the result presented in this section in no way requires that Petri nets be used to specify the system.

The problem with the construction in [Ultes-Nitsche (1999)] is that any pair of actions $(a, b) \in \Delta$ not satisfying $h(a) = h(b)$ must be discarded from $\Delta$. Taking this observation into the context of constructing persistent sets, we see that for a set $P \subseteq En(s)$ to be persistent at $s$, we now have the additional requirement that

for all $a \in P$, and for all actions $b$ that occur in words in $cont(s, L) \cap (\Sigma - P)^*$, we must have $h(a) = h(b)$.

A moment's thought tells us that this in turn implies that all actions in $P$ and all actions in $cont(s, L) \cap (\Sigma - P)^*$ must *all* have the same image under $h$ which is unlikely to occur in most real-life examples. Therefore, the "persistent sets" at most states will probably be trivial, and the reduction obtained either minor or possibly non-existent.

The first task of my PhD was to make this result more practical, starting from the consideration that the requirement that pairs $(a, b) \in \Delta$ must satisfy $h(a) = h(b)$ is in fact too strong for our needs. Recall that this '$h$-compatibility' restriction was imposed on $\Delta$ to ensure that all words in the same trace had identical images under abstraction, so that we could discard some words in the same trace without losing essential information on the abstract level . In fact, it is fairly clear that to ensure this we need only stipulate that pairs $(a, b) \in \Delta$ satisfy $h(ab) = h(ba)$ [proved later on in Lemma 3.8], which opens up the extra possibilities where one or more of $a$ and $b$ may be hidden by $h$. Since 'good' abstractions hide many actions, this allows for a much expanded independence relation compared with that in [Ultes-Nitsche (1999)].

Relatively soon, we managed to extend [Ultes-Nitsche (1999)] to give the desired result for the improved independence relation, but with the restriction [Ultes-Nitsche and St James (2000)] that non-trivial persistent sets $P \subseteq En(s)$ must contain a non-hidden action, say $a \in P$ with $h(a) = A \neq \varepsilon$. Unfortunately, this caused a similar but less severe problem to that in the case of [Ultes-Nitsche (1999)]; that is, that all actions $a$ contained in $P$ or occurring in words in $cont(s, L) \cap (\Sigma - P)^*$ must satisfy $h(a) \in \{\varepsilon, A\}$, which is still a restriction that can severely impact the level of reduction that we may obtain . Eventually, though, this restriction was removed and the concept of completely hidden persistent sets became a possibility. Such sets are extremely desirable because all actions in a completely hidden set are automatically $h$-compatible with any other actions, so the restriction of $h$-compatibility ceases to be a problem in this case.

## 3.2 Statement of the Result

We now give a more formal description of this result, and a proof that it is correct. Prior to this, though, we need some definitions.

### 3.2.1 Trace Reductions

We present here the definition of *trace reductions* (taken from [Godefroid (1991, 1995)]), together with the abstraction-related modifications we need to make to the original definition of independence.

**Definition 3.1.** Let $L \subseteq \Sigma^*$ be a regular language, $a, b \in \Sigma$, $w \in L$. Then $a$ and $b$ are *$h$-compatibly independent* at $w \in L$ if and only if

i) $a$ and $b$ are independent at $w \in L$ according to Definition 2.19; and

ii) $h(ab) = h(ba)$

**Definition 3.2.** Let $L \subseteq \Sigma^*$ be a regular language, $\Delta \subseteq \Sigma \times \Sigma \times L$. Then $\Delta$ is an *$h$-compatible independence relation over $L$* if and only if $(a, b, w)$ implies that $a$ and $b$ are $h$-compatibly independent at $w$.

**Definition 3.3.** A *trace automaton*, $\mathcal{A}_L^\Delta$ [Godefroid (1991, 1995)], is an automaton resulting from a modified form of the persistent set selective search (Algorithm 2.2) on $L$ using the independence relation $\Delta$. The modification comes from changing the function "Persistent_Set($s$)" in line 7 to "Persistent_Set_Satisfying_Proviso($s$)" ; this new function returns a set $P \subseteq En(s)$ that satisfies:

i) $P$ is persistent at $s$; and

ii) If $P$ is non-trivial (i.e. $P \subset En(s)$) then there is a state $s'$ and an action $a \in P$ such that $s' \notin stack$ and $s \xrightarrow{a} s'$

The second condition is the *proviso* in "Persistent_Set_Satisfying_Proviso($s$)", and the reason why it is required probably seems a little opaque at this point; in brief, it is a "fairness" condition that ensures that no actions are unfairly "ignored" during the construction of the trace reduction: please refer to Section B.2 for more details. Whether or not a candidate persistent set satisfies this proviso is obviously easily decidable.

**Definition 3.4.** We call the language represented by such a trace automaton $\mathcal{A}_L^{\Delta}$ a *trace reduction* of $L$ with respect to $\Delta$, and denote it by $R_L^{\Delta}$.

It was shown in [Godefroid (1991, 1995)] that:

**Theorem 3.5.** *If $R_L^{\Delta}$ is a trace reduction of the language $L$, then:*

*i)* $R_L^{\Delta} \subseteq L$;

*(i.e. $R_L^{\Delta}$ is no more expressive than $L$. In fact, the trace automaton is a sub-automaton of the automaton representing $L$)*

*ii)* $R_L^{\Delta}$ *is prefix-closed when $L$ is (as we are assuming throughout this document); and*

*iii)* $\forall w' \in R_L^{\Delta}, \forall v \in cont(w, L), \exists w'' \in cont(w', R_L^{\Delta}), v \in pre\left([w'']_{\Delta,w'}\right)$.

*(i.e. for all states $s$ in the trace automaton representing $R_L^{\Delta}$, and for all words in the continuation from $s$ in the automaton representing $L$, this word appears (in a slightly tangled form) in the continuation from $s$ in the automaton representing $R_L^{\Delta}$.)*

The algorithm, then is as shown in Algorithm 3.1. It is taken from [Godefroid (1991, 1995); Godefroid and Wolper (1993); Wolper and Godefroid (1991)] with small modifications, such as our addition of "#" labels at deadlock states, as described on Page 8.

We say that $R_L^{\Delta}$ is $h$-compatible if and only if $\Delta$ is, and assume henceforth that this is the case. The main result we are working towards can be expressed as in Theorem 3.6.

**Theorem 3.6.** *If $h(R_L^{\Delta})$ does not contain maximal words, then the condition*

$$\lim(h(R_L^{\Delta})) \models_{WF} \mathcal{P} \Leftrightarrow \lim(L) \models_{WF} \Leftrightarrow h^{-1}(\mathcal{P})$$

*holds if and only if $h$ is weakly continuation-closed on $R_L^{\Delta}$.*

## 3.3 Proof of Correctness

We now begin to prove the result of Theorem 3.6, beginning with a few preliminary Lemmas.

**Lemma 3.7** (trace-equivalent words lead to the same state). *Let $w \in L$, $w', w'' \in cont(w, L)$ such that $w' \equiv_{\Delta,w} w''$. Then $cont(ww', L) = cont(ww'', L)$.*

*Proof.* By definition, $w' \equiv_{\Delta,w} w''$ implies that $w'$ can be transformed into $w''$ by repeatedly swapping pairs of adjacent actions. So let $(w^i)_{1\leq i\leq n}$ be a sequence of words

```
$Q \leftarrow \{s_0\}$ ; $stack \leftarrow \phi$;
**push** $(s_0)$ onto $stack$;
**sub** DFS()
    $s = top(stack)$;
    $Q \leftarrow Q \cup \{s\}$;
    **let** $T$ =Persistent_Set_Satisfying_Proviso($s$);
    **if** $T = \phi$ **then**          // The state $s$ must be a deadlock
        $\delta \leftarrow \delta \cup (s, \#, s)$;  // state. Add a loop to itself marked
    **endif**                     // with the dummy label, "#"
    **for** each action $a$ in $T${
        find state $s'$ such that firing $a$ at $s$ leads to $s'$;
        $\delta \leftarrow \delta \cup (s, a, s')$;
        **if** $s' \notin Q$
            **push** $(s')$ onto $stack$;
            **call** DFS();
        **endif**
    }
    **pop** $s$ from $stack$;
**end sub**
```

**Algorithm 3.1** – The Algorithm for Constructing a Trace Reduction

such that $w^1 = w'$ and $w^n = w''$ and for all $1 \leq i < n-1$, $w^{i+1}$ can be obtained from $w^i$ by transposing precisely one pair of independent actions adjacent in $w^i$. Let $a$ and $b$ be that pair; then $w^i$ may be written as $w^i = uabv$, $w^{i+1}$ as $w^{i+1} = ubav$ for some $u, v \in \Sigma^*$. Because $a$ and $b$ are independent at $wu$, we have from part i) of Definition 2.19 that $cont(wuab, L) = cont(wuba, L)$ and as an immediate consequence that $cont(wuabv, L) = cont(wubav, L)$. Thus, $cont(ww^i, L) = cont(ww^{i+1}, L)$ for each $1 \leq i < n-1$, so $cont(ww', L) = cont(ww'', L)$. □

**Lemma 3.8** (all words in the same trace have the same abstract image). *Let $w \in L$, $w', w'' \in cont(w, L)$ such that $w' \equiv_{\Delta,w} w''$. Then $h(w') = h(w'')$.*

*Proof.* By definition, $w' \equiv_{\Delta,w} w''$ implies that $w'$ can be transformed into $w''$ by repeatedly swapping pairs of adjacent actions. So let $(w^i)_{1 \leq i \leq n}$ be a sequence of words such that $w^1 = w'$ and $w^n = w''$ and for all $1 \leq i < n-1$, $w^{i+1}$ can be obtained from $w^i$ by transposing precisely one pair of independent actions adjacent in $w^i$. Let $a$, $b$ be that pair ; then, since by assumption $\Delta$ is $h$-compatible, $h(ab) = h(ba)$. Thus $h(w^{i+1}) = h(w^i)$ for each $1 \leq i < n-1$, and so $h(w') = h(w'')$. □

**Lemma 3.9** (equivalence of continuations under abstraction).

$$\forall w' \in R_L^{\Delta}, \quad h(cont(w', R_L^{\Delta})) = h(cont(w', L)).$$

*Proof.* Theorem 3.5i) implies that $cont(w', R_L^\Delta) \subseteq cont(w', L)$, so after applying $h$ to both sides we see that it is sufficient to prove that $h(cont(w', L)) \subseteq h(cont(w', R_L^\Delta))$.

Let $w \in cont(w', L)$; then by Theorem 3.5iii), we have that $\exists w'' \in cont(w', R_L^\Delta)$ such that $w \in pre\left([w'']_{\Delta,w'}\right)$. So let $v$ be such that $wv \equiv_{\Delta,w'} w''$; then by Lemma 3.8, $h(wv) = h(w'')$. Since $h$ is non-length-increasing [as noted after the definition of abstracting homomorphisms], and $R_L^\Delta$ is prefix-closed by Theorem 3.5ii), there is some prefix $u$ of $w''$ such that $h(u) = h(w)$, and further, $u \in cont(w', R_L^\Delta)$. Thus, for all $w \in cont(w', L)$ there is $u \in cont(w', R_L^\Delta)$ such that $h(u) = h(w)$; so $h(cont(w', L)) \subseteq h(cont(w', R_L^\Delta))$, and the lemma is proved. □

**Corollary 3.10.**

$$h(L) = h(R_L^\Delta)$$

*Proof.* Follows immediately from Lemma 3.9 upon setting $w' = \varepsilon$. □

**Lemma 3.11** (one "half" of continuation-closure is guaranteed).

$$\forall w \in L, h(cont(w, L)) \subseteq cont(h(w), h(L))$$

*Proof.* Let $v \in cont(w, L)$. Then $wv \in L$. So $h(wv) = h(w)h(v) \in h(L)$. Therefore, $h(v) \in cont(h(w), h(L))$; hence result. □

**Lemma 3.12** ("associativity" of continuations).

$$cont(uv, cont(w, L)) = cont(v, cont(wu, L))$$

*Proof.*

$$\begin{aligned}
cont(uv, cont(w, L)) &= \{x | uvx \in cont(w, L)\} \\
&= \{x | uvx \in \{y | wy \in L\}\} \\
&= \{x | wuvx \in L\} \\
&= \{x | vx \in \{y | wuy \in L\}\} \\
&= \{x | vx \in cont(wu, L)\} \\
&= cont(v, cont(wu, L))
\end{aligned}$$

□

**Lemma 3.13.** *If $h$ is weakly continuation-closed on $R_L^\Delta$, then $h$ is weakly continuation-closed on $L$.*

*Proof.* We wish to show that for any $w \in L$, there exists $r \in cont(h(w), h(L))$ such that

$$cont(r, cont(h(w), h(L))) = cont(r, h(cont(w, L))) \tag{3.1}$$

By Lemma 3.11, it suffices to show that the LHS of (3.1) is contained in the RHS of (3.1).

Let $w \in L$. Then there exists $w'' \in R_L^{\Delta}$ (and so $w'' \in L$ also, by Theorem 3.5i)) such that $w \in pre\left([w'']_{\Delta,\epsilon}\right)$ by Theorem 3.5iii). Therefore, there exists $v$ such that $wv \equiv_{\Delta,\epsilon} w''$; since $w'' \in L$, $wv \in L$ as well. Since $\Delta$ is an $h$-compatible independence relation, we observe that $cont(wv, L) = cont(w'', L)$ (Lemma 3.7) and

$$h(wv) = h(w'') \tag{3.2}$$

Applying $h$ to both sides of $cont(wv, L) = cont(w'', L)$ and invoking the result of Lemma 3.9 leads to the string of equations

$$h(cont(wv, L)) = h(cont(w'', L)) = h(cont(w'', R_L^{\Delta})) \tag{3.3}$$

Because $h$ is weakly continuation-closed on $R_L^{\Delta}$, there exists $u \in cont(h(w), h(R_L^{\Delta}))$ such that

$$cont(u, cont(h(w''), h(R_L^{\Delta}))) = cont(u, h(cont(w'', R_L^{\Delta}))) \tag{3.4}$$

Now consider the set of equations

$$\begin{aligned}
cont(h(v)u, cont(h(w), h(L)) &= cont(u, cont(h(w)h(v), h(L))) \text{ by Lemma 3.12} \\
&= cont(u, cont(h(w''), h(R_L^{\Delta}))) \text{ by definition of } h, \\
&\quad (3.2), \text{ and Corollary 3.10} \\
&= cont(u, h(cont(w'', R_L^{\Delta}))) \text{ by (3.4)} \\
&= cont(u, h(cont(wv, L))) \text{ by (3.3)} \\
&= cont(u, h(cont(v, cont(w, L)))) \text{ by Lemma 3.12} \\
&\subseteq cont(u, cont(h(v), h(cont(w, L)))) \text{ by Lemma 3.11} \\
&= cont(h(v)u, h(cont(w, L))) \text{ by Lemma 3.12}
\end{aligned}$$

Thus

$$cont(h(v)u, cont(h(w), h(L))) \subseteq cont(h(v)u, h(cont(w, L)))$$

Setting $r = h(v)u \in cont(h(w), h(L))$ and comparing this with our objective (3.1) proves the result. □

**Lemma 3.14.** *If $h$ is weakly continuation-closed on $L$, then $h$ is weakly continuation-closed on $R_L^{\Delta}$.*

*Proof.* Let $w' \in R_L^{\Delta}$. Then $w' \in L$ by Theorem 3.5i). Using the fact that $h$ is weakly continuation-closed on $L$, let $u \in cont(h(w), h(L))$ be such that

$$cont(u, cont(h(w'), h(L))) = cont(u, h(cont(w', L)))$$

Then using this result in conjunction with Corollary 3.10 and Lemma 3.9 we have that

$$\begin{aligned} cont(u, cont(h(w'), h(R_L^{\Delta}))) &= cont(u, cont(h(w'), h(L))) \\ &= cont(u, h(cont(w', L))) \\ &= cont(u, h(cont(w', R_L^{\Delta}))) \end{aligned}$$

□

The proof of the Main Theorem of this thesis, Theorem 3.6, then follows immediately from Theorem 2.18, Corollary 3.10 and Lemmas 3.13 & 3.14.

The restriction to languages not containing maximal words is again circumvented by using the '#' action, as described on page 8 . So this result improves upon that in [Ultes-Nitsche (1999)] by expanding the independence relation, which as a rule of thumb increases the magnitude of the state-space reduction obtained via a trace reduction[2]. It should also be noted that the 'better' the abstracting homomorphism is, in the rather subjective sense described in the section on abstraction, the closer still the $h-$compatible independence relation is to the original independence relation, and again the amount of reduction obtained should be further increased. In systems in which a relatively high proportion of pairs of transitions are independent, for example *weakly interacting systems*, partial-order methods (without the encumbrance of $h$-compatibility) often give a reduction of orders of magnitude, so this, in principle at least, is a very promising result.

## 3.4 A Small Example

We now give a small and rather artificial example of the usage of this theorem. The example is much too small to give an idea of the power of the result (although even in this case, the reduction is not entirely insignificant) and is given solely as an attempt to clarify the concepts involved.

The dummy example $\mathcal{P}$ (shown in Figure 3.1; note that the place numbers have been omitted for clarity) is loosely based on the idea of two copies of the earlier example Petri net (Figure 2.1) running concurrently.

[2] The intuition is that the more actions are independent of each other, the more likely a given pair are independent, which in turn means that smaller subsets of actions at states are more likely to be persistent, and so the reduction in the size of the automaton due to partial-order methods is increased.

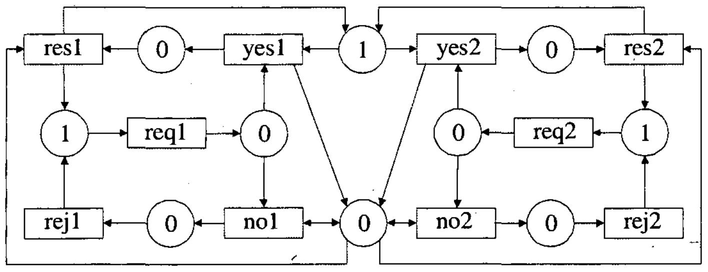


FIGURE 3.1: An Example Petri Net Specification, $\mathcal{P}$.

Although Theorem 3.6 tells us that there is no need to, we have calculated the state-space of the system (i.e. the automaton representing the behaviour, lim($L$), of $\mathcal{P}$). It has 14 states and 28 transitions, and is shown in Figure 3.2.

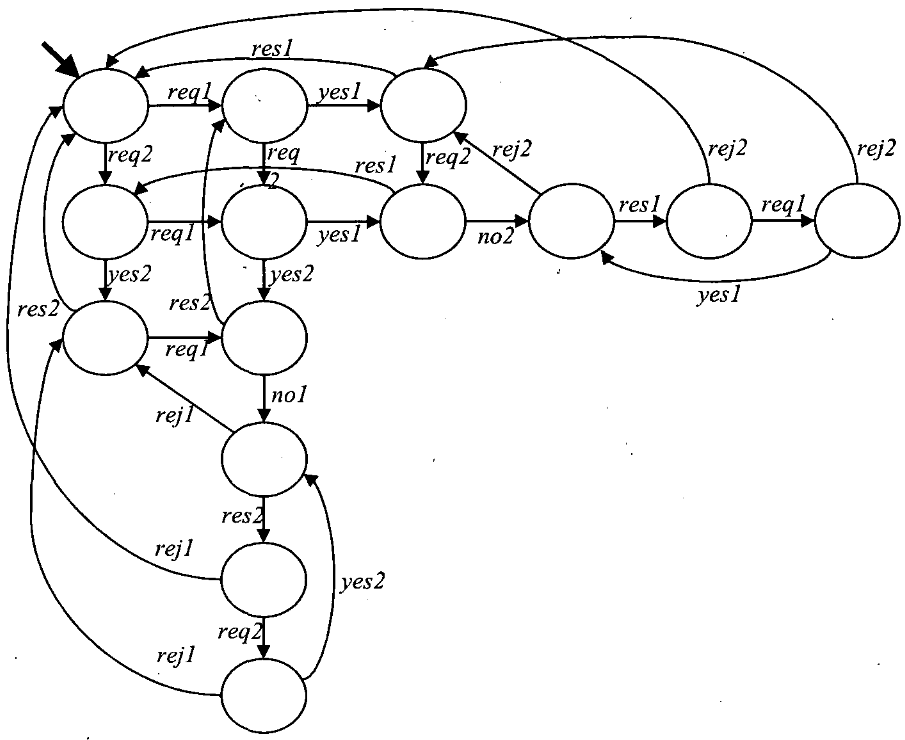


FIGURE 3.2: The behaviour of the system in Figure 3.1.

We set the abstracting homomorphism $h$ to be the identity on the set of actions *req1*, *res1* and *rej1*, and the empty word on all other actions. We form the $h$-compatible independence relation by first constructing the ordinary independence relation (consisting of all pairs of actions that do not share a place) and then refining it by removing all pairs that are not $h$-compatible; the result is shown in Table 3.1. Note that this table shows that, as mentioned earlier, an independence relation need not be reflexive nor transitive: *res*1 and many other actions are clearly not independent of themselves (as they disable themselves when fired), and *req*1 is independent of *req*2, *req*2 is independent of *res*1, but *req*1 is *not* independent of *res*1.

| | *res1* | *rej1* | *req1* | *yes1* | *no1* | *res2* | *rej2* | *req2* | *yes2* | *no2* |
|---|---|---|---|---|---|---|---|---|---|---|
| *res1* | | | | | | | X | X | | |
| *rej1* | | | | | | X | X | X | X | X |
| *req1* | | | | | | X | X | X | X | X |
| *yes1* | | | | | | | X | X | | |
| *no1* | | | | | | | X | X | | |
| *res2* | | X | X | | | | | | | |
| *rej2* | X | X | X | X | X | | | | | |
| *req2* | X | X | X | X | X | | | | | |
| *yes2* | | X | X | | | | | | | |
| *no2* | | X | X | | | | | | | |

TABLE 3.1: Table representing the independence relation, $\Delta$, for the system in Figure 3.1

We then constructed the trace automaton representing the trace reduction of $L$, $R_L^\Delta$, using an implementation of the trace reduction algorithm (Figure 3.1) in a preliminary version of our Petri Net tool that works only with low-level Petri nets. The persistent sets were calculated at each state of the trace automaton using a generic persistent set calculator, tailored for use with low-level nets. In Chapter 4, we describe how to construct persistent sets for "richer" classes of Petri Nets. The resulting trace automaton is shown in Figure 3.3; it has 10 states and 16 transitions.

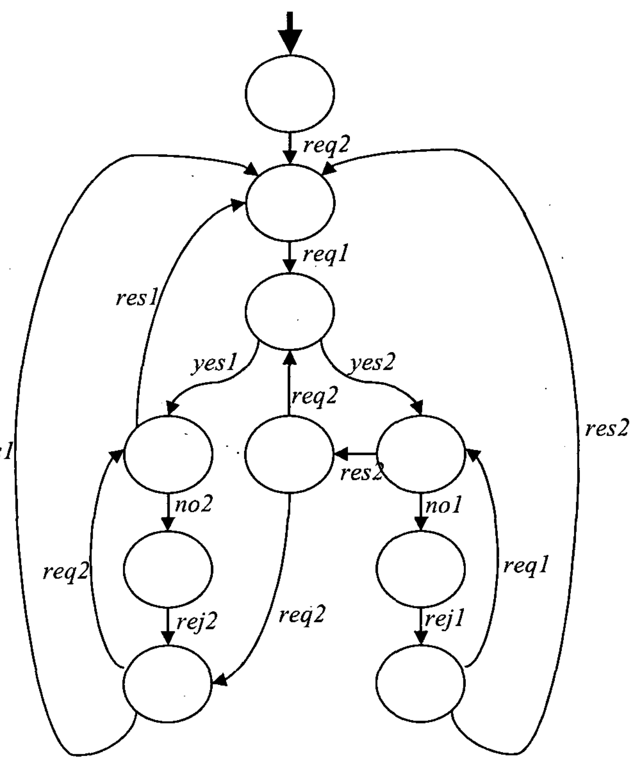


FIGURE 3.3: The automaton representing the trace reduction of the system

Using a manual application of the result of Appendix A (after noting that the set of all states but the first one form a strongly-connected bottom component of the automaton), it can be shown that $h$ is WCC on $R_L^{\triangle}$ (and hence on $L$, from Lemma 3.13) which according to Theorem 3.6, implies that we may deduce properties of the full behaviour from the abstraction of $R_L^{\triangle}$. Computing this w.r.t. $h$ gives the automaton shown in Figure 3.4. As mentioned in our comparison of behaviour abstraction and data abstraction, the relationship between concrete and abstract states under behaviour abstraction is not particularly meaningful or useful, so we will not show the relation here.

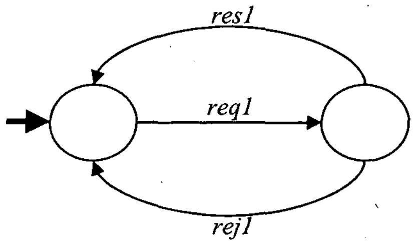


FIGURE 3.4: The abstract behaviour of the dummy system.

From this abstract automaton, and using the result of Theorem 3.6, we see that the full behaviour of the system must satisfy, for example, the property $G(\mathit{req1} \Rightarrow F(\mathit{res1}))$ ("it is always the case that firing the action *req1* will eventually give a *res1*") within fairness.

## 3.5 Sleep-sets

The sleep-set technique of Godefroid and Wolper [Godefroid (1995)] is a commonly used supplement to the persistent-set selective search designed to further reduce the number of *transitions* (*not* states) in the "trace" automaton. Sleep sets are not especially relevant to our results, so we will give only a vague description of them. We feel, though, that they warrant at least a brief mention for two reasons; firstly, it should be noted that this commonly-used optimization for the persistent-set selective search is in fact *not* compatible with the main result of this thesis; and secondly, because it may be the case that it can be used to improve a result presented in our section on compositional verification; see Appendix B. As we will see, a persistent-set selective search augmented by the sleep-set technique results, in general, in a greatly weakened version of a trace automaton that does not allow us to decide weak continuation closure of the original language based on the trace reduction.

Even when using the persistent-set selective search, which dramatically reduces the number of interleavings of independent actions considered, some easily-avoided redundant interleavings will still be considered during construction; for example, in the case where a persistent set contains several mutually independent actions. The sleep-set technique is geared towards removing some of the actions followed while guaranteeing that we do not 'miss out' on any states by doing so.

The simplest example occurs when two independent actions $a, b$ occur at a persistent set at $s$, as shown in Figure 3.5.

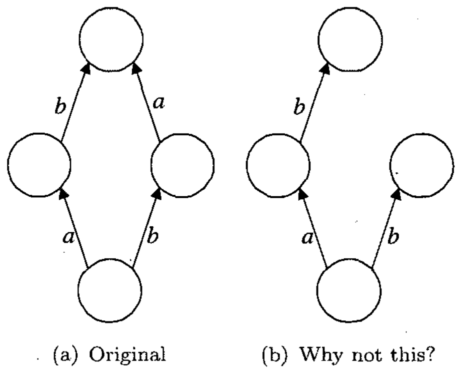


FIGURE 3.5: The simplest situation where the sleep-set technique may be of use.

A more dramatic example occurs when the same three mutually independent actions occur in successive persistent sets, as in Figure 3.6.

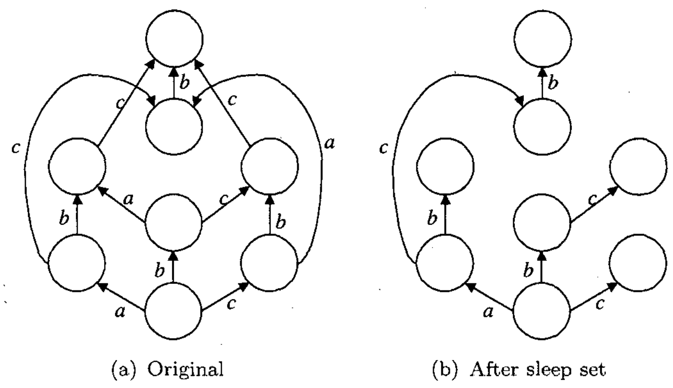


FIGURE 3.6: A More Complicated Sleep-set Example.

Briefly, the sleep-set technique achieves this reduction on-the-fly by maintaining for each state $s$ a set called a "sleep-set" . This is a set $s.Sleep \subseteq \Sigma$ of actions that need not be followed from $s$ since it can be guaranteed (using reasons similar to the result of Lemma 3.7) that doing so will not lead us to any states that we could not reach by not following these actions from $s$ (for more details, see [Godefroid (1995)]). For example, if $a$ and $b$ are independent at $s$ and are both followed from $s$ (so that both words $ab$ and $ba$ can be followed from $s$) we may pick one of these two words to follow: say, $ab$. At the state $s'$, say, reached from following $b$, we would add $a$ to the sleep-set at $s'$ as a signal that $a$ does not need to be followed from $s'$: if $a$ and $b$ are independent at $s$ and we are following $ab$ from $s$, we do not gain much from following $ba$.

If the construction of $R_L^\Delta$ incorporates the sleep-set technique, then Theorem 3.5iii) holds only at states with an empty sleep set; i.e. Theorem 3.5iii) becomes "$\forall w \in L$

such that $s_0 \xrightarrow{w} s'$ with $s'.Sleep = \phi \ldots$", so we do not have a trace reduction. This is because, as one can see from the examples above, sleep-sets generate a large amount of dead-end states, or at the very least severely disrupt the original sets $cont(s, R_L^\Delta)$. Thus, the results of Lemma 3.9 cannot be guaranteed to hold and in fact there are examples in which the blind application of the sleep-set technique contravenes the result of Lemma 3.9 and the main result, Theorem 3.6, as we will see in the next sub-section.

However, we will show later that the sleep-set technique preserves sufficient information to be useful in a compositional verification result presented later (see Chapter 5 and Appendix B for more details).

### 3.5.1 Sleep-sets and Theorem 3.6

In this sub-section, we give an example of a language $L$, an abstracting homomorphism $h$ on $L$, and a trace reduction $R_L^\Delta$ of $L$ augmented by the sleep-set technique that does *not* satisfy the statement $h$ is WCC on $L$ if and only if $h$ is WCC on $R_L^\Delta$.

Consider the low-level Petri Net (as shown in Figure 3.7, with set of actions $\Sigma = \{a, b, c, d, e\}$). Note that $a$ and $b$ are independent (we can tell this from a static analysis of the Petri net: they share access to no places, and so cannot affect one another).

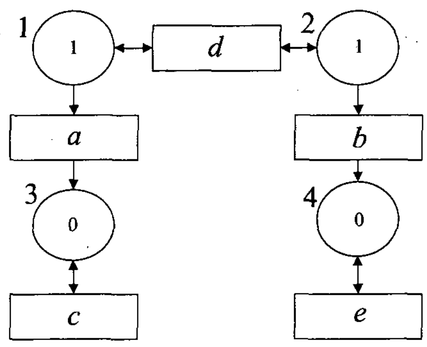


FIGURE 3.7: The Petri Net specification we will use for the counter-example

Expand it to give the language $L$, as represented by the automaton in Figure 3.8.

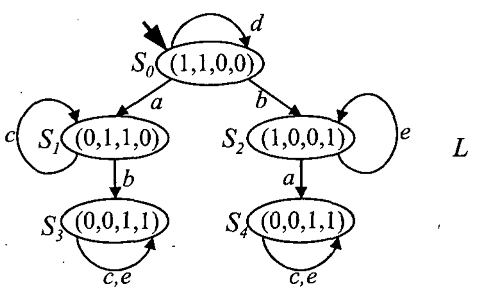


FIGURE 3.8: The language $L$ accepted by the Petri Net of Figure 3.7

Define an abstracting homomorphism $h : \Sigma \rightarrow \Sigma' \cup \{\varepsilon\}$ with $\Sigma' = \{B, C\}$ as follows:

$$h(a) = h(d) = h(e) = \varepsilon,$$
$$h(b) = B, h(c) = C$$

The abstraction $h(L)$ may then be represented by the automaton in Figure 3.9.

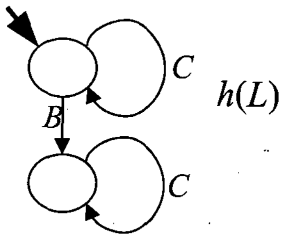


FIGURE 3.9: The abstraction of $L$

We now construct a partial-order reduced version of $L$, $R_L$, as follows: at the initial state, we follow the trivial persistent set. After we follow $a$, we may add $a$ to the sleep-set at $S_2$ as, as we explained earlier, the sleep-set technique rests on the assumption that if we are going to follow $ab$, there is no point in following $ba$ [when $a$ and $b$ are independent]. At every state but $S_2$, we follow the trivial persistent set and do not use sleep-sets. At $S_2$, we follow the trivial set minus the $S_2.sleep = \{a\}$, as the sleep-set technique says we may. We end up with the automaton as shown in Figure 3.10.

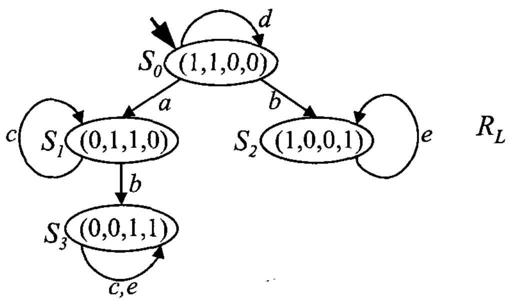


FIGURE 3.10: The representation of the sleep-set reduced version of $L$

It is now easy to show that $h$ is weakly continuation-closed on $L$; to see this, see that $L$ can be represented by the automaton in Figure 3.11 (which is actually just a minimized version of the automaton in Figure 3.8).

A manual application of the result of Appendix A shows that $h$ is weakly continuation-closed on $L$. However, $h$ is *not* weakly-continuation-closed on $R_L$; to see this, let $w = b$ : then $h(cont(w, L)) = \varepsilon$, and $cont(h(w), h(L)) = cont(B, h(L)) = C^*$. If $h$ were weakly continuation-closed on $L$, then there would be some $v \in cont(h(w), h(L)) = C^*$ such

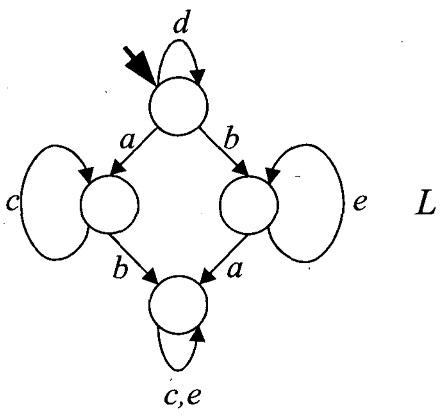


FIGURE 3.11: The minimized version of the automaton representing $L$

that

$$\begin{aligned} cont(v, h(cont(w, L)) &= cont(v, cont(h(w), h(L)); \text{ i.e.} \\ cont(v, \varepsilon) &= cont(v, C^*); \text{ i.e.} \\ \varepsilon &= C^* \end{aligned}$$

which is plainly a contradiction. Thus we cannot use the unmodified sleep-set technique with our result. It is possible that some restricted version of the sleep-set technique may be invented to complement this result, but we currently have no idea what form this new technique might take.

## 3.6 Summary

We have improved the practicality of the result of [Nitsche and Ochsenschläger (1996)] (Theorem 2.18 in this thesis) by showing that constructing the trace reduction $R_L^\Delta$ of a regular, prefix-closed language $L$, with an independence relation $\Delta$ modified so as to be $h$-compatible, enables us to compute the abstract behaviour $h(L)$ and also to determine whether $h$ is WCC on $L$, all without having to construct the behaviour of $L$. Since the necessity of constructing this full behaviour was the main impediment to the usefulness of the result of Theorem 2.18, this has the potential to be a very useful result. Interestingly, however, the result is in fact *not* compatible with the sleep-set technique, which is often used to further decrease the amount of transitions in the trace automaton.

Our result improves the practicality of [Ultes-Nitsche (1999)] by loosening what the restrictions for two independent actions $a$ and $b$ to be $h$-compatible; formerly, it was that $h(a) = h(b)$, which is a very strong restriction that would likely make the existence of non-trivial persistent sets very rare. The result of [Ultes-Nitsche and St James (2000)] weakened the restriction so that $a$ and $b$ could be called $h$-compatible if $h(ab) = h(ba)$, which opened up the possibility of one of $a$ and $b$ being hidden, but unfortunately also required that any persistent set must contain at least one non-hidden action, which again made non-trivial persistent sets comparatively rare. The result presented in this chapter removes this restriction on persistent sets, allowing such sets to be completely hidden by the abstraction. This has the effect of making non-trivial persistent sets much more likely to be found. A practical example of the usage of this result is given in Chapter 6.

While the main result of this chapter is applicable to any means of specification that can be unfolded into a LTS, we have used Petri Net-specified systems to illustrate the concepts, as algorithms for constructing persistent sets for Petri-net specified systems are well understood. The new requirement of $h$-compatibility of the independence relation and resulting persistent sets, however, requires us to re-visit these algorithms, which will be the subject of the next chapter.

# Chapter 4

# High-level Petri Nets and Persistent Sets

## 4.1 Overview

Generally, the construction of a persistent set at a state in a Petri net specified system is reasonably straightforward: one need only construct a *stubborn set*[Valmari (1991a,b)] and intersect it with the set of actions currently enabled at that state[Godefroid (1995)]. The result of Chapter 3, however, requires a modification to the definition of independence, and hence to persistent sets: the independence relation must now be *abstraction-compatible*, which may mean that the old stubborn set methods must be adjusted. For this reason, and also the fact that some problems with Petri nets (notably the *binding problem*) are of importance in the following chapter and the fact that our chosen practical example of Chapter 6 would be difficult to construct using low-level Petri nets, this chapter will explore high-level Petri nets and the stubborn-set approach to constructing persistent sets in high-level Petri net specified systems.

Specifying systems using low-level Petri nets is a cumbersome chore, and for this reason high-level Petri nets are more commonly used; high-level Petri nets bear approximately the same relation to low-level nets as high-level languages do to low-level ones, such as assembly code. As with high-level languages, high-level Petri nets come in many different varieties and dialects, and we present in this Chapter a brief overview of the dialect used to generate the practical results presented in Chapter 6 (an implementation of a variant of the Sliding Windows Protocol). Since this practical example benefits from the addition of "native" Petri net queues, we describe also the queue extension to our Petri net dialect and show how they fit into the definition of independence. We then go on to describe the stubborn set technique for constructing persistent sets, and how we may slightly relax the definition of independence when constructing stubborn sets, and then explore how the requirement of abstraction-compatibility complicates matters. Finally, we attempt to put everything together and provide some heuristics for constructing reasonably good stubborn sets while taking into account abstraction-compatibility and making use of some special properties of queues.

## 4.2 High-level Petri Nets

A rigorous formulation of our conception of Petri nets would be very long and detailed and, for the purposes of this chapter, quite unnecessary. We will restrict ourselves, then, to a relatively vague formulation; usually, in the areas where we have given insufficient detail as to how something works or is defined, "common-sense" is usually enough to flesh-out the details.

The most important differences between high- and low-level Petri nets are as follows. Perhaps the most important difference is that high-level Petri nets allow tokens/place contents to belong to user-defined data-types that may be more complex than the set of positive integers that low-level nets are restricted to. Usually, these data-types are constructed (informally) from the built-in data types (finite ranges of the natural numbers, Boolean) and user-specified enumerated sets, using operators such as the Cartesian product, unions, and Kleene star (although, as we will see later, our formulation does not use Kleene stars). Each place has a data-type associated with it (which we call the *domain* of the place), and a token may only be placed in a place if it belongs to the domain of the place. The *contents* of a place, which could only be a non-negative integral number of tokens in the low-level case, is now a *multi-set* of tokens drawn from the domain of the place. As before, the "state" of a high-level Petri net is determined by the contents of its places.

The actions of a high-level Petri net differ from those in a low-level one in several ways. An action in our high-level Petri nets conception consists of a *base-action* $a_{base} \in \Sigma_{base}$ together with a *binding* of the variables of $a_{base}$; each base-action $a_{base} \in \Sigma_{base}$ has associated with it a set of variables $x_1^a, x_2^a, ...x_{m^a}^a$, and a binding of $a_{base}$ is simply an assignment $x_1^a = v_1, x_2^a = v_2, ...x_{m^a}^a = v_{m^a}$ of values to these variables. The set of actions, then, is the set consisting of all pairs of base-actions and bindings to their variables, and is denoted $\Sigma$. When constructing the automaton representing the behaviour of the Petri net, each transition is labelled with the base-action followed and the variable binding with which it was fired; i.e. the input alphabet is $\Sigma$, consistent with earlier chapters. Also, each action now has a *guard* predicate associated with it; a little more about this later.

Base-actions still have input/ output arcs, but each of these now carries an *inscription*, which is a multi-set of *expressions* in the set of variables associated with $a$[1]. *Expressions*, informally speaking, are built from variables, user-defined constants, user-defined

[1] The *guard* of an action $a$ is also an expression in the set of variables of $a$.

functions, the *arithmetical operators* +, -, / (integer division), * (multiplication) and mod ("remainder" function); formation into *vectors* (e.g. $(1,3,x)$) and some special in-built functions: the *projection* function, $proj(x,n)$, which returns the $n$th component of vector $x$ (if it exists) (e.g. $proj((1,3),2) = 3$, $proj((1,(2,3)),2) = (2,3)$, etc.); the *isin* function, $isin\ (x,D)$ which returns true or false as $x$ belongs to the domain $D$; and the *if* function, $if(testExpression, expr1, expr2)$, which returns the result of *expr1* if the *testExpression* evaluates to *true*, and *expr2* otherwise. User-defined functions consist of the function name, a list of arguments, and a return expression. User-defined functions, constants and domains are usually declared in a special section of the Petri net called the *preamble.* There is an additional restriction on the arc inscriptions of a base-action $a$; each variable associated with $a$ must occur "naked" in the arc inscription of an input arc of $a$. This restriction will be described more fully in Section 4.2.1.

We begin now to present a slightly more formal definition of high-level Petri nets, starting with the definitions of some of the basic building blocks required; those of *multi-sets, domains* and *tokens.*

**Definition 4.1.** A *multi-set* $MS(S)$ over the set $S$ is defined as a function

$$MS(S) : S \rightarrow \mathbb{N}$$

Let $S$ be a set, $A : S \rightarrow \mathbb{N}$ and $B : S \rightarrow \mathbb{N}$ be multi-sets over $S$. Then we say $A \subseteq B$ if and only if $A(s) \leq B(s)$ for all $s \in S$. We define the multi-set $A + B$ over $S$ to be the multi-set over $S$ defined by $(A + B)(s) = A(s) + B(s)$ for all $s \in S$. If $A \subseteq B$, then we define $B - A$ as the mapping $(B - A)(s) = B(s) - A(s)$ for all $s \in S$. Note that addition of multi-sets is commutative.

**Definition 4.2.** A *domain* is any set that can be constructed inductively according to the following rules.

- For any $a, b \in \mathbb{N}$ with $a \leqslant b$, the finite range $\{a, a+1, ..., b\}$, represented as $[a, b]$ is a domain.
- Any set $\{s_1, s_2, ...s_n\}$ of strings is a domain.
- If $D_1$ and $D_2$ are domains, then the union $D_1 \cup D_2$ is a domain.
- If $D_1, D_2, ..., D_n$ are domains, then the set $\{(d_1, d_2, .., d_n)\, |d_i \in D_i\}$ is a domain.

We define $D$ as the set of all possible domains.

**Definition 4.3.** Let $D$ be the set of all domains, where domains are defined as in Definition 4.2. The the set of all possible tokens is equal to $\bigcup_{d \in D} d \cup \{null\}$, where *null* is called the *null token.* The null token is defined to belong to all domains, and has some other special properties that will be described later.

We define $T$ to be the set of all possible tokens.

**Definition 4.4.** A *token multi set* is a multi-set over the set $T - \{null\}$.

As mentioned earlier, the null token *null* has some special properties, which are as follows: for any token multi-set $M$ we define $null \in M$ and $M + null = M$ and $M - null = M$.

**Definition 4.5.** A *token expression in n variables* is a mapping $expr : T^n \longrightarrow T$.

Define $Expr(n)$ to be the set of all expressions in $n$ variables, and $Expr = \bigcup_{n=1,2,..} Expr(n)$ to be the set of all expressions in any number of variables.

**Definition 4.6.** A *high-level Petri net* $\mathcal{P}$ is a vector $(\Sigma_{base}, P, I, M_0)$ where

- $\Sigma_{base} = \{a_1, a_2, ..., a_n\}$ is the set of *base actions.* Each base-action $a \in \Sigma_{base}$ is associated with a set $\{x_1^a, x_2^a, ..., x_{m_a}^a\}$ of variables and a *guard expression guard*$(a)$ which is an expression in the $m_a$ variables associated with $a$;
- $P = \{p_1, p_2, ..., p_r\}$ is the set of *places.* Each place $p_i \in P$ has an associated domain $D_i \in D$;
- $I$ is a partial function $I : \Sigma_{base} \times P \vee P \times \Sigma_{base} \longrightarrow MS\,(Expr)$ representing the *arcs* between places and base-actions. If $I$ is defined for the pair $(p_i, a)$ or $(a, p_i)$, then the result must be an expression in $m_a$ variables, where $m_a$ is the number of variable associated with the base-action $a$; and
- $M_0 : P \longrightarrow MS(D)$ is the *initial marking* or the *initial state* of the Petri net $\mathcal{P}$. States are defined a little more formally in Definition 4.7.

**Definition 4.7.** A *marking* (or, as we prefer, a *state*) $S$ of a Petri net $\mathcal{P} = (\Sigma_{base}, P, I, M_0)$ is a mapping $S : P \longrightarrow MS(D)$ which maps the place $p_i$ to a multi-set over the domain $D_i$ of $p_i$. This multi-set is called the *contents* of the place $p_i$ at the state $S$.

**Definition 4.8.** An *action* $A = (a, (v_1, v_2, ..., v_{m^a}))$ in a high-level Petri net $\mathcal{P} = (\Sigma_{base}, P, I, M_0)$ is a pair consisting of a base-action $a \in \Sigma_{base}$ together with a binding of token values to the variables associated with $a$ e.g. $x_1^a = v_1, x_2^a = v_2, ...x_{m^a}^a = v_{m^a}$, $v_i \in T$.

**Definition 4.9.** Let $\mathcal{P} = (\Sigma_{base}, P, I, M_0)$ be a Petri net, $a_{base} \in \Sigma_{base}$ be a base-action. The the set $in\,(a_{base}) \subseteq P$ of *input places* of $a_{base}$ is equal to the set of places $p_i \in P$ such that $I((p_i, a_{base}))$ is defined and yields a non-empty multi-set. Similary, the set *out* $(a_{base}) \subseteq P$ of *output places* of $a_{base}$ is equal to the set of places $p_i \in P$ such that $I((a_{base}, p_i))$ is defined and yields a non-empty set.

Let $A$ be the action consisting of the base-action $a$ and the binding $x_1^a = v_1, x_2^a = v_2, ...x_{m^a}^a = v_{m^a}$, $v_i \in T$ of its variables. Let $p_{in} \in in\,(a)$ be an input place. Then the *multi-set of tokens taken from $p_{in}$ by $A$* is defined as $I((p_{in}, a))(v_1, v_2, ...v_{m_a})$ and abbreviated *remove*$(A, p)$. If the token $t$ is in this multi-set of tokens taken from $p_{in}$ by $A$, then we say that *$A$ takes $t$ from $p_{in}$*.

Let $p_{out} \in out\,(a_{base}) \subseteq P$ be an output place of $a$, and $A$ an action consisting of the base-action $a$ and the binding $x_1^a = v_1, x_2^a = v_2, ...x_{m^a}^a = v_{m^a}$. Then the *multi-set of tokens added to $p$ by $A$* is defined as $I((a, p_{out}))(v_1, v_2, ...v_{m_a})$ and abbreviated *add*$(A, p)$. If the token $t$ is in this multi-set of tokens taken added to $p_{out}$ by $A$, then we say that *$A$ adds $t$ to $p_{out}$*.

**Definition 4.10.** Let $\mathcal{P} = (\Sigma_{base}, P, I, M_0)$ be a Petri net, $S : P \rightarrow MS(D)$ be a state that maps each $p_i$ to a multi-set over $D_i$, the domain associated with $p_i$. Let $A$ be the action consisting of the base-action $a$ and the binding $x_1^a = v_1, x_2^a = v_2, ...x_{m^a}^a = v_{m^a}$, $v_i \in T$ of its variables. Then we say that $A$ is *enabled* at the state $S$ if and only if it passes all of the following tests.

i) If *guard*$(a)(v_1, v_2, ...v_{m_a})$ is equal to 0 or *null*, then $A$ is not enabled at $S$.

ii) For each $p_{in} \in in(a)$, if $A$ takes a non-null token $t$ from $p_{in}$ that is not in the domain of $p_{in}$, then $A$ is not enabled at $S$.

iii) For each $p_{out} \in out(a)$, if $A$ takes a non-null token $t$ from $p_{out}$ that is not in the domain of $p_{out}$, then $A$ is not enabled at $S$.

Note: if $A$ fails any of the above three tests, then $A$ is said to be an *invalid action*, and can never be enabled at any state. Actions that pass these three checks are deemed *valid*, although there is still the possibilty that a valid action will never be enabled at any state.

iv) For each $p_{in} \in in(a)$, the multi-set $I((p, a))(v_1, v_2, ...v_{m_a})$ of tokens removed from $p_{in}$ must be contained in $p_{in}$; i.e. if it is not the case that $I((p, a))(v_1, v_2, ...v_{m_a}) \leqslant S(p_{in})$, then $A$ is not enabled at $S$.

If $A$ passes all of these tests, then $A$ is enabled at $S$. If $A$ is enabled at $S$, we also say that $A$ may *fire* at $S$.

**Definition 4.11.** Let $\mathcal{P} = (\Sigma_{base}, P, I, M_0)$ be a Petri net, $S : P \rightarrow MS(D)$ be a state that maps each $p_i$ to a multi-set over its associated domain $D_i$. Let $A$ be the action consisting of the base-action $a$ and the binding $x_1^a = v_1, x_2^a = v_2, ...x_{m^a}^a = v_{m^a}$, $v_i \in T$ of its variables that may fire at $S$. Then *the state $S'$ reached by firing $A$ at $S$* is defined as follows.

- If $p \in P$ is in $in\,(a)$, then $S'(p) = S(p) - I((p, a))(v_1, v_2, ...v_{m_a})$.
- If $p \in P$ is in $out\,(a)$, then $S'(p) = S(p) + I((p, a))(v_1, v_2, ...v_{m_a})$.
- Otherwise, $S'(p) = S(p)$.

### 4.2.1 A Problem with Bindings

In order to correctly compute $En(s)$ for a given state $s$ we must compute, for each base-action $a$, the set $B$ of bindings to the variables of $a$ with which $a$ may fire at $s$.

**Definition 4.12.** Let $a$ be a base-action and $s$ be a state of a Petri-net $\mathcal{P}$. Let $B : \{x_1^a, x_2^a, ..., x_{m^a}^a\} \rightarrow T$ be any binding to the variables of $a$ such that $a$ may fire with binding $B$ at $s$. Then $B$ is called a *valid binding* for $a$ at $s$.

The problem of generating, for each base-action $a$ at each state $s$, a *finite* set of bindings such that no valid binding for $a$ at $s$ is overlooked(a pre-requisite for computing $En(s)$) is worth mentioning since it causes special problems in the area of compositional verification (indeed, as we will see, one idea for compositional verification renders the problem insoluble). The basic problem is exemplified by the example Petri net specification of Figure 4.1. The set of variables associated with the action $a$ is $\{x\}$.

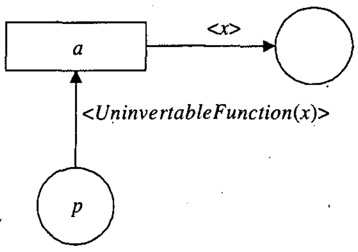


FIGURE 4.1: A situation that causes binding problems.

It is obvious in this example from the conditions under which $a$ may fire with a binding that the set of bindings $x = v$ with which $a$ may fire is precisely the set such that the token returned by $UninvertableFunction(v)$ is in the contents of place $p$ i.e. $UninvertableFunction\,(v) \in \{1, 2, 3\}$.[2]

The function $UninvertableFunction\,(x)$ represents a function for which the problem of constructing, for some value $A$, the set $\{x|UninvertableFunction\,(x) = A\}$ is not feasible; perhaps because there is no known algorithm that effects the inversion of the function, or because this set is infinite, etc. We could arbitrarily restrict $x$ to lie in some finite set, but then it is, in general, impossible to prove that we are not missing out any valid bindings of $x$. So one can see that in general, the problem of finding a finite set of bindings that demonstrably includes all valid bindings is not solvable.

The most commonly used method to avoid this is to restrict the set of acceptable Petri nets by stipulating that for each base action $a \in \Sigma_{base}$, and for each variable $x$ associated with $a$, $x$ occurs "naked" in some expression of some inscription arc of $a$ [Burkhardt et al.

[2] This illustrates an important general principle; in the general case, virtually the only way to ensure that the set of bindings is finite is by using the contents of input places to restrict the bindings considered. Ideally, for the sake of efficiency, we should generate only *valid* bindings, but in general, some instantly recognisably invalid bindings will almost always be proposed and have to be checked.

(1989)]. We say that $x$ occurs naked in an expression if and only if there is some sequence of projections that can be applied to the expression that results in the variable $x$: as we will see, this allows us to determine all values of $x$ which could possibly be part of a valid binding, effectively "solving" the arc inscription for $x$. For example, the variable $x$ occurs naked in the following expressions:

- $expr =< x >$ [Empty sequence of projections]
- $expr =< (x, y) >$ $[proj(expr, 1) = x]$
- $expr =< (y + 3, (x, z)) >$ $[proj(proj(expr, 2), 1) = x]$
- $expr =< (f(y), (z, (x, y))) >$ $[proj(proj(proj(expr, 2), 2), 1) = x]$

but not in

$< f(x) >$, $< x + 3 >$ or $< (x + 2, 4) >$.[3]

It can be shown then that if $P$ is the function that effects the sequence of projections that is applied to the expression in the input arc $I$ in which $x$ occurs naked, then for $a$ to fire at $s$ with a binding that assigns $x = v$, the input place $p$ connected to $I$ must contain a token $t$ such that $P(t) = v$. In fact, we can invert this to use the set of tokens in $p$ to get a set of bindings for $x$ that will not 'miss' any valid bindings for $x$. Here is an example that will hopefully clarify this; we wish to generate a "sufficient" set of bindings for the variable $x$ for the base-action in Figure 4.2.

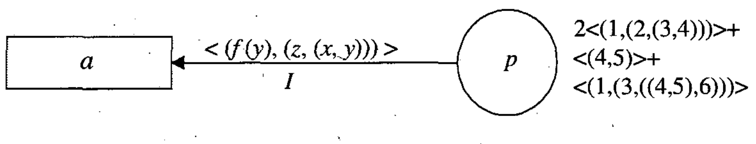


FIGURE 4.2: An example of deriving a sufficient set of bindings.

The function $P$, from earlier, is defined by $P(t) = proj(proj(proj(t, 2), 2), 1)$. Applying $P$ to each token $t$ in $p$ gives the sufficient set of bindings for $x$ as $x = 3$ and $x = (4, 5)$. No other bindings of $x$ need be considered. We use the same technique to get sufficient bindings for all of the other associated variables of $a$, and so get a set of sufficient bindings for *all* variables associated with $a$ by letting each variable range independently over all of its sufficient bindings. As mentioned in Footnote 2, this will likely generate many "invalid" bindings, but this is really unavoidable, and unimportant anyway, since invalid bindings will be eliminated by the firing conditions of Definition 4.10.

[3] Note that this restriction is a little too restrictive; for example, in the latter two examples, the possible bindings of $x$ can be easily found because the "function"$x+3$ is very easily inverted. Nevertheless, for the sake of simplicity, this is the restriction we will adopt.

## 4.3 High-level Petri Nets and Persistent Sets

Having given a brief description of high-level Petri nets and their semantics, we now turn our attention to the problem of computing a persistent set at a given state in a high-level Petri net and see how the additional requirements of $h$-compatibility imposed by the main result of this thesis may be accommodated. The problem reduces to two components; finding an independence relation for the actions in the high-level net, and the actual algorithm for taking this relation and using it to produce a persistent set. We deal with the problem of independence first, and show that the definition of independence is actually a little too strong for our needs when constructing persistent sets.

### 4.3.1 High-level Petri Nets and Independence

We review the definition of persistent sets:

**Definition 4.13.** Given a language $L$ over an alphabet $\Sigma$, and a word $w \in L$, the set $P \subseteq En(w)$ (where $En(w)$ is the set of actions enabled "at $w$"; i.e. $En(w) = \Sigma \cap cont(w, L)$) is persistent at $w$ if and only if $P$ is non-empty and for all $p \in P$ and all words $v = a_1a_2...a_n \in cont(w, L) \cap (\Sigma - P)^*$, for all $i = 2, ..., n-1$ we have that $p$ is independent of $a_i$ at $wa_1a_2...a_{i-1}$.

We draw attention to the statement "$p$ must be independent of $a_i$ at $wa_1a_2..a_{i-1}$". Now, $p$ is enabled at $w$, and none of the $a_i$'s may disable it my definition of a peristent set; thus, $p$ is still enabled at $wa_1a_2...a_{i-1}$. By assumption, $a_i$ is also enabled at $wa_1a_2...a_{i-1}$, and so we need pay no heed to whether $a_i$ and $p$ may enable each other at $wa_1a_2...a_{i-1}$ - they are both certainly enabled! That is, we may ignore the "actions may not enable one another" condition of Definition 2.19 when constructing persistent sets, giving us a "weaker" notion of independence which we will call *weak-independence*:

**Definition 4.14.** Given a language $L$ over an alphabet $\Sigma$, and a word $w \in L$, two actions $a$ and $b$ enabled at $w$ are said to be *weakly-independent* at $w$ if and only if:

i) $\forall w \in L, cont(wab, L) = cont(wba, L)$

ii) $\forall w \in L,\ (a, b) \in cont(w, L)$ implies $ab, ba \in cont(w, L)$.

The second item simply asserts that $a$ and $b$ may not disable each other at $w$. As before, if two actions $a$ and $b$ are weakly independent at all $w \in L$, we simply say that they are *weakly-independent*.This is contrasted with the original definition of independence, which included the additional restriction that independent actions could not enable one another at a state.

Analogously, this leads to the notion of a weak-independence relation:

**Definition 4.15.** A *weak independence relation* $\Delta$ over a language $L \subseteq \Sigma^*$ is a relation $\Delta \subseteq \Sigma \times \Sigma$ such that $(a, b) \in \Delta$ implies that $a$ and $b$ are weakly-independent.

It is beneficial, when constructing persistent sets, to use in place of $\Delta$ a "weakened" version of $\Delta$, which we will call $\Delta_{weak}$, defined in Definition 4.16.

**Definition 4.16.** Let $L \subseteq \Sigma^*$ be a language over the alphabet $\Sigma$, $\Delta \subseteq \Sigma \times \Sigma \times L$. Then the *weak-independence analogue* of $\Delta$, $\Delta_{weak}$, is the weak independence relation satisfying the following property: $\forall w \in L$, if $a$ and $b$ may not enable one another at $w \in L$, then $(a, b, w) \in \Delta$.

The motivation for this definition is obtaining a weak-independence relation from a full independence relation $\Delta$ by adding pairs of actions that are *only* not allowed in $\Delta$ because they may enable one another[4].

We prove now that we can use this weaker version of independence to simplify the construction of persistent sets; the nature of the simplification won't be made clear until the section on the stubborn set technique, so for now, we present it as the following theorem:

**Theorem 4.17.** *$L \subseteq \Sigma^*$ be a language over an alphabet $\Sigma$, $w$ be a word in $L$, and $P \subseteq En(w)$ be a non-empty subset of the set of enabled actions at $w$ in $L$. Let $\Delta \subseteq \Sigma \times \Sigma \times L$ be an independence relation over $L$, and $\Delta_{weak} \subseteq \Sigma \times \Sigma$ be the weakened version of $\Delta$.*

*If for all $p \in P$ and actions $a$ that appear in words in $cont(w, L) \cap (\Sigma - P)^*$ we have that $(a, p) \in \Delta_{weak}$, then for all $p \in P$, $v = a_1a_2...a_n \in (\Sigma - P)^* \cap cont(w, L)$ we have, for each $m = 1, 2, ..., n$, that $(p, a_m, wa_1a_2..a_{m-1}) \in \Delta$; i.e. $P$ is persistent at $w$.*

*Proof.* Let $p$ be any action in $P$ and $v = a_1a_2...a_n$ be any word in $cont(w, L) \cap (\Sigma - P)^*$. By assumption, $(p, a_i) \in \Delta_{weak}$ for all $i = 1, 2, ..., n$. Now, $p$ must be enabled at $wa_1a_2...a_{m-1}$ for each $m = 1, 2, ...n + 1$; else, let $i \geq 0$ be the smallest number such that $p$ is not enabled at $wa_1a_2...a_i$. Then $i$ cannot be zero, as $p$ is enabled at $w$, so $p$ is enabled at $wa_1a_2...a_{i-1}$ by definition of $i$. But then $a_i$ must disable $p$ at $wa_1a_2...a_{i-1}$, contradicting the fact that $\Delta_{weak}$ is a weak-independence relation.

So both $p$, $a_m$ must be enabled at $wa_1a_2...a_{m-1}$ for each $m = 1, 2, ..., n + 1$, so $p$ and $a_m$ (trivially) cannot enable one another at $wa_1a_2...a_{m-1}$. Combining this with the fact that $(p, a_m) \in \Delta_{weak}$ by assumption, we have that $(p, a_m, wa_1a_2...a_{m-1}) \in \Delta$ from Definition 4.16. □

[4]The "only" is stressed as there are other reasons why one might wish to remove a triple $(a, b, w)$ from an independence relation e.g. for $h$-compatibility. We do not want to include pairs of actions in our weakened version of $\Delta$ if they are disqualified from inclusion in $\Delta$ for other reasons besides non-enablement.

So, given a high-level Petri net, how does one go about constructing a weak independence relation?

Note that our conception of an action, which now encapsulates both the base action and, importantly, the binding of values to its variables ensures that two things remain constant from state to state: the result of the guard expression, and the multi-sets resulting from evaluating the arc inscriptions on the input/ output places of the base-action (and hence, the set of tokens removed from each input place and added to each output place when the action fires)[5].

This tells us two things: firstly, that the only way an action $a$ can disable another action $b$ is by taking one of the tokens required by $b$ from the input place for $b$ (as $a$ cannot affect the result of evaluating $b$'s guard expression, which is a constant); and secondly, that any pair of actions always "commute" i.e. that following $ab$ and $ba$ at a state $s$ (if both words are followable from $s$) will lead to the same state[6]. Therefore:

**Theorem 4.18.** *If two actions $a, b \in \Sigma$ do not remove the same token from the same place, then $a$ and $b$ are weakly-independent at* all *states.*

It is possible for two actions that remove the same token $t$ from the same place $p$ (and which would ordinarily be assumed to be dependent, for safety) to be weakly-independent at a state $s$ if the place $p$ in the state $s$ contains sufficiently many copies of token $t$ for $a$ to fire while leaving enough $t$'s in $p$ for $b$ to fire and vice-versa, but we ignore this apparent optimisation as in practice exploiting it is difficult and in fact gives very little in the way of gain [personal correspondence with Pierre Wolper]. The reason it initially seemed like a worthwhile optimisation is that it seemed that we could exploit it in construction of a persistent set $s'$ if, for example, $a$ was in a candidate set $P$ and we could show that at all states $s$ reachable by following a word in $(\Sigma - P)^* \cap cont(s', L)$ from $s'$ such that $b$ is enabled at $s$, $a$ and $b$ would be weakly-independent at $s$. However, verifying this would involve exploring all such states $s$ which would be a very expensive procedure, so for safety we simply assume that $a$ and $b$ are always dependent. . Thus, we restrict ourselves to the "uniform" (invariant from state to state) weak independence relation implied by Theorem 4.18.

Before moving on to show how to use the weak-independence relation to construct persistent sets, we will briefly discuss the incorporation of *queues* into our Petri net framework and show how to alter the independence relation in order to take the addition of queues into account.

[5]It was this second fact that enabled us to define the functions $add(a, p)$ and $remove(a, p)$ in Definition 4.9.

[6]Informally, this follows from the definitions of the place contents of the successor states after following an action (Definition 4.11) coupled with the properties of addition/ subtraction of multi-sets (e.g. addition of multi-sets is commutative)

### 4.3.2 Queues and Independence

When modelling communications networks with Petri nets, it is beneficial to have some sort of *queue* structure to simulate messages being sent down a channel. Originally, we were planning to use a place whose domain was the Kleene star of an element type and whose contents took the form of a concatenated string representing a queue of that element type, but as we will see in Section 4.3.4, our choice of implementation enforces finite (and preferably small) domains for places, which rules this approach out. Cyclic queues may be modelled in the framework presented earlier with a little ingenuity, but are "inefficient" in the sense that they swell the size of the state-space immensely. Eventually, we decided to add a new, native type of place which can be used as a queue. A queue $q$, where each element is a member of the domain $d$ and which can hold $maxinqueue$ elements, is identified by writing $Queue(d, maxinqueue)$ as the domain of the place. If $maxinqueue$ is equal to zero, the queue $q$ is *unbounded.* The contents of a place that is a queue $q$ over domain $D_q$ is an element in $D_q^{maxinqueue}$ if $maxinqueue > 0$, and $D_q^*$ if $maxinqueue = 0$ i.e. if $q$ is unbounded; the definition of states should be modified to reflect this. If the contents of $q$ at a state $s$ is equal to $d_1 d_2 ... d_n$, $d_i \in D_q$, then $d_1$ is called the *tail* of $q$ at $s$ and $d_n$ is called the *head* or *leading element* of $q$ at $s$.

The changes to the Petri net semantics brought about by the addition of queues are as follows; the changes are fairly minor, so we give only a brief, informal description.

If an the base action of an action $a$ has no input/ output places that are queues, the conditions for whether $a$ may fire are unchanged.

Otherwise, a few new conditions are added. Let $a$ be an action, and let

$$q_{out} = Queue(d, maxinqueue)$$

be an output place of the base action of $a$ which is also a queue. Then in addition to the rules for deciding whether $a$ may fire presented earlier, we add the following.

- $a$ may not fire if $add(a, q_{out})$ is non-null and $add(a, q_{out})$ is not a single token ($t$, say).
- $a$ may not fire if $add(a, q_{out}) = t$, a non-null token, and $q_{out}$ contains $maxinqueue$ elements i.e. we cannot add a non-null token to a full queue.

Now we consider the restrictions imposed when $a$ has an input place

$$q_{in} = Queue(d, maxinqueue)$$

- $a$ may not fire if $add(a, q_{in})$ is non-null and $add(a, q_{in})$ is not a single token ($t$, say).

- $a$ may not fire if $add(a, q_{in}) = t$, a non-null token, but $t$ is not the element at the head of $q_{in}$ (which obviously precludes the case where $q_{in}$ is empty).

If $a$ meets all the requirements for firing, then the changes to the output places that are not queues are the same as always. For each output place $q_{out}$ that is a queue, $q_{out}$ either stays the same if $t = add(a, q_{out})$ is the null token, or $t$ is appended to the tail of $q_{out}$ i.e. if $contents(q_{out})$ represents the current contents of $q_{out}$, then the new contents of $q_{out}$ is given by $t.contents(q_{out})$. For each input place $q_{in}$, $q_{in}$ stays the same if $t = add(a, q_{out})$ is null, and has the leading element removed if $t$ is non-null i.e. f $contents(q_{out}) = d_1 d_2 ... d_n$ represents the current contents of $q_{out}$, then the new contents of $q_{out}$ is given by $d_1 d_2 ... d_{n-1}$.

How does this new semantics affect the weak-independence relation described in Theorem 4.18? It actually remains largely unchanged; the only new dependencies arise between actions that add to the same queue.

Assume two actions $a$ and $b$ add different (non-null) tokens $t_a$ and $t_b$ to the same place, $q$, and that $q = Queue(d, maxinqueue)$ is a queue. Let $s$ be a state such that the words $ab$ and $ba$ are each followable from $s$. Then the contents of $q$ in the state $s'$ reached by following $ab$ from $s$ would begin with $t_b t_a$ i.e. would be of the form $t_b t_a.contents(q, s)$. Following $ba$ from $s$, however, would leave $q$ with the string $t_a t_b.contents(q, s)$; thus the sequences of actions $ab$ and $ba$ lead to different states, contravening the definition of independence (weak or otherwise). Thus, a pair of actions that add different tokens to the same queue are not weakly-independent.

In fact, even if $a$ and $b$ added the same non-null token $t$ to the queue $q$ (in which case $ab$ and $ba$ would indeed lead to the same state when followed from $s$), then $a$ and $b$ will still probably not be weakly-independent, for $a$ may disable $b$ by filling up the queue (so that $q$ contains $maxinqueue$ elements) and preventing it from firing. Even if $q$ in the current state $s$ had ample space remaining, we cannot guarantee that the words occurring "outside" of a candidate persistent set $P \subseteq En(s)$ containing $a$ at $s$ (i.e. those words in $cont(s, L) \cap (\Sigma - P)^*$) do not fill up $q$, disabling $a$; thus, it is best, when constructing persistent sets, to assume that two actions $a$ and $b$ that add non-null tokens to the same queue $q$ are not weakly-independent.

**Theorem 4.19.** *If two actions $a, b \in \Sigma$ do not remove the same token from the same place, and do not both add non-null tokens to the same queue, then $a$ and $b$ are weakly-independent at* all *states.*

We now move on to the basic method for computing persistent sets: Valmari's *stubborn set technique*[Valmari (1991a,b)]. After presenting this technique, we describe a few heuristics intended to aid the efficient construction of reasonably small persistent sets.

### 4.3.3 The Stubborn Set Technique

Constructing persistent sets is not a simple task, and although we have seen one method of construction (see the example involving Figure 2.7), this particular method simply does not scale to non-trivial systems: as noted in that example, it involves a potentially large amount of exploration at each state for multiple subsets of the set of enabled actions at each state. Valmari's *Stubborn Set Technique* provides a much more effective means of computing persistent sets. The intuition is that, if we can ensure for a given subset $P$ of the set of enabled actions $En(s)$ at a state $s$ that each action dependent on any action in $P$ is either a) also in $P$; or b) cannot fire "outside of $P$" (i.e. does not occur in any word in $(\Sigma - P)^* \cap cont(s, L)$), then $P$ will be persistent. The stubborn set technique initially starts with a set *StubbornSet* consisting of a single action in $En(s)$ and then begins to progressively add actions to *StubbornSet* until the subset of actions $P = En(s) \cap StubbornSet$ is persistent at $s$. If an action $b$ occurs in a word in $(\Sigma - P)^* \cap cont(s, L)$ and $b$ is dependent on some action in $P$, then the stubborn set technique will add actions to $P$ in order to prevent $b$ from appearing in $(\Sigma - P)^* \cap cont(s, L)$ by determining which actions are required to fire in order to enable $b$, and attempting to prevent *these* actions from appearing in $(\Sigma - P)^* \cap cont(s, L)$, recursively. The process continues until there are no such $b$, at which point $P$ will be a persistent set.

A more detailed description follows. Throughout this section, we will deem two actions $a$ and $b$ to be *dependent* if the pair $(a, b)$ is not contained in the weak-independence relation $\Delta_{weak}$ described in Theorem 4.18.

Assume that we wish to find a persistent set $P$ at the state $s$. We pick an action $a_1 \in En(s)$, and attempt to form a persistent set around it. The stubborn set is initially empty; we now place $a_1$ in it. The first step in constructing a persistent set containing $a_1$ is to find all actions $b$ that are dependent on $a_1$ - it is very important to note that this includes actions *which may not currently be enabled.* For each such $b$ that *is* enabled at $s$, this action $b$ must be added to the stubborn set, as it eventually must be contained in any persistent set containing $a_1$. For each such action $b$ that is *not* currently enabled, the situation is more complicated; if there were a sequence of actions outside of our desired persistent set[7] that enables $b$, then our persistent set could be invalid; hence, we must make sure that $b$ can *not* occur outside of the persistent set. We do this by picking a set of actions (call it $NES(b, s)$) such that $b$ cannot fire from the current state $s$ without an action in $NES(b, s)$ having fired first, and add these to the stubborn set. Then we pick another set of actions actions such that none of the currently disabled actions in $NES(b, s)$ can fire without one of these actions firing first, and add these to the stubborn set, etc. We continue this "chain" until we have added sufficient actions to ensure that $b$ cannot fire outside of the eventual persistent set. Then we repeat this

[7]By "outside of our persistent set", we mean a sequence of actions that can occur in $cont(s, L) \cap (\Sigma - P)^*$

process for each enabled action in the stubborn set that has not yet been dealt with in this manner, until no new actions are added to the stubborn set.

This describes the basic idea behind the construction, but the algorithm is actually structured slightly differently; it can be expressed as in Algorithm 4.1.

In this algorithm, *StubbornSetNew* is a set that represents actions that must be processed. Eventually, all actions in here will be transferred to *StubbornSet*. Initially, both of the sets *StubbornSetNew* and *StubbornSet* are empty; then an action is chosen from $a \in En(s)$ and added to *StubbornSetNew*. The main loop then begins; at each iteration, an action $a$ is removed from *StubbornSetNew*. There are two cases; the first case occurs when $a \in En(s)$ (Line 17). Since $a$ will be transferred to *StubbornSet*, and since the final persistent set is equal to *StubbornSet* $\cap$ $En(s)$ (Line 25), $a$ will be in the final persistent set, $P$. Therefore, we cannot allow any actions dependent on $a$ (including actions that are *not* enabled at $s$) to occur outside of $P$; therefore, all such actions are added to *StubbornSetNew* for processing (Lines 17–21). Note that we never add an action to *StubbornSetNew* that is already there or that is in *StubbornSet*; such actions have already been processed (or will soon be processed), and there is no need to process them again.

The second case is when $a$ is not enabled at $s$; we see then that $a$ is either dependent on some action that will be in the final persistent set, $P$, or is part of a chain of actions that are required to enable some action dependent on an action that will be in $P$; in either case, $a$ must be prevented from firing outside $P$, so we find a set of actions that are required to enable $a$[8], and these actions to *StubbornSetNew* (Lines 8–11).

The stubborn set construction ends when there are no new actions left to process (i.e. when *StubbornSetNew* $= \phi$), at which point we compute the final persistent set (Line 25). Note that the choice of the initial action to place in the stubborn set and the subsequent choices of deficient place and token pairs used to generate necessary enabling sets for actions are non-deterministic, and the size of the eventual persistent set and the number of steps required to construct it are dependent on the quality of the heuristics employed for making these choices. We will briefly mention some heuristics for making these choices later on this chapter.

A quick definition is useful in formulating the algorithm:

**Definition 4.20.** Let $L \subseteq \Sigma^*$ be a language over an alphabet $\Sigma$, let $w \in L$, and let $a \in \Sigma$ be an action not enabled at $w$. Then a *necessary enabling set for a at w* is a set of actions $NES(a, w) \subseteq \Sigma$ such that for any $v = a_1a_2...a_na \in cont(w, L)$, at least one of the $a_i$'s must be in $NES(a, w)$, with an analogous definition in terms of states.

In other words, a necessary enabling set $NES(a, s)$ for a disabled action $a$ at a state $s$

[8]For a more formal definition of what we mean here, see Definition 4.20.

```
1:  $StubbornSet \leftarrow \phi, StubbornSetNew \leftarrow \phi$;
2:  pick $a_1 \in En(s)$, and add it to $StubbornSetNew$;
3:  repeat
4:      pick $a \in StubbornSetNew$;
5:      $StubbornSetNew \leftarrow StubbornSetNew \setminus a$;
6:      if $a \notin En(s)$   /* This action is one that must be prevented
7:                              from firing outside of our persistent set */
8:          find a necessary enabling set $NES(a, s)$ for $a$ at $s$;
9:          for each $c \in NES(a, s)$
10:             if $c \notin StubbornSet \cup NewStubbornSet$
11:                 $StubbornSetNew \leftarrow StubbornSetNew \cup \{c\}$;
12:             endif
13:         next
14:     else    /* Add this action to the stubborn set
15:                and all actions dependent on it to StubbornSetNew
16:                for processing */
17:         for all $b$ such that $(a, b) \notin \Delta_{weak}$/* Includes $b$ not enabled at $s$ */
18:             if $b \notin StubbornSet \cup NewStubbornSet$
19:                 $StubbornSetNew \leftarrow StubbornSetNew \cup \{b\}$;
20:             endif
21:         next
22:     endif
23:     $StubbornSet \leftarrow StubbornSet \cup \{a\}$;
24: until $StubbornSetNew = \phi$;
25: $PersistentSet(s) = StubbornSet(s) \cap En(s)$;
```

**Algorithm 4.1** – The Basic Stubborn Set Algorithm

is a set of actions such that any occurrence of $a$ in a word followable from $s$ must be preceded by at least one of the actions in $NES(a, s)$.

We now comment on the main stubborn set algorithm, Algorithm 4.1, and discuss how to obtain an efficient implementation of the algorithm.

## 4.3.4 Towards an Efficient Petri Net Implementation

In this section, we present some fairly obvious and well-known observations that help with the efficient construction of stubborn sets (although the work involving queues is, as far as we are aware, novel). We begin by ignoring the complications caused by the requirement of $h$-compatibility, which we will explore in Section 4.3.4.2.

#### 4.3.4.1 Without Abstraction Compatibility

We draw attention to how Definition 4.16 coupled with Theorem 4.17 help with computing stubborn sets: given an enabled action $a$ which must be included in the final persistent set, we can instantly decide which other actions $b$ must be added to the stubborn set in order to ensure that the final set of enabled actions in the stubborn set is persistent; the set of actions is equal to[9] the set $\{b|(a,b) \notin \Delta_{weak}\}$, as shown in Lines 17–21. Computing the set of actions that would have to be added straight from the "raw" definitions (Definitions 2.23/ 2.24) would be most difficult, and would probably require us to explore sub-automata from $s$ to ensure that the final candidate persistent set was in fact persistent; this would, of course, be very inefficient indeed.

The problem of computing the set of all actions dependent on an action $a$, and finding a necessary enabling set for an action $a$ both require a certain task to be performed, which we describe now. For a given action $a$, the problem of computing the set $\{b|(a,b) \notin \Delta_{weak}\}$ is solved with the help of Theorem 4.19; we simply need to find *all* actions that remove the same token $t$ from the same place $p$, and if $a$ adds a non-null token to a queue $q$, *all* actions that add a non-null token to the same queue. When we come to consider $h$-compatibility, we may, if $h(a) \neq \epsilon$, also need to be able to find all actions $b$ for which $h(ab) \neq h(ba)$.

For computing a necessary enabling set for an action, we need to perform a similar task. Before describing this task, we need some further analysis and terminology.

Say an action $a$ is not enabled at a state $s$. We know from earlier that the guard condition for a given action is a constant, so if the guard condition for $a$ is false, it can never be enabled and so can be ignored. Similarly, as the arc inscriptions do not vary, if $a$ attempts to add/ remove a token $t$ from a place $p$ which is not in the domain of $p$, then $a$ can never fire and can again be ignored. So from the semantics presented earlier, we know that either $a$ attempts to remove more copies of a token $t$ from a place $p$ than are contained in $p$ at $s$, or it attempts to add a non-null token to a queue $q$ which is full at $s$.

**Definition 4.21.** Let $a$ be a valid action that is not enabled at a state $s$. If $a$ attempts to remove a token $t$ from a place $p$, and $p$ does not contain $t$ at $s$, then $p$ is called a *deficient place* for $a$ at $s$ (or simply a deficient place if $a$ and $s$ are understood). The pair $(p,t)$ is called a *deficient place and token pair* for $a$ at $s$ (or simply a deficient place and token pair).

If $a$ attempts to place a non-null token $t$ in a queue $q$ and $q$ is full at $s$, then $q$ is called a *blocking queue* for $a$ at $s$, or simply a blocking queue.

[9]Or more accurately, is a subset of; this method could well force actions that are not required for persistence to be added to the set. This does not seem to be too much of a problem, fortunately.

The fact that if there is a deficient place and token pair $(p, t)$ for an action $a$ at $s$, then $a$ cannot fire until some action places the token $t$ in the place $p$, gives a strong hint as to how a set $NES(a, s)$ can be constructed; in fact, we can just pick $NES(a, s)$ equal to the set of all actions that place the token $t$ in the place $p$[10]. Similarly, if $a$ has a blocking queue $q$ at $s$, a necessary enabling set for $a$ could be the set of all actions that remove a non-null token from queue, thus unblocking it.

So in order to perform either of these two operations (finding all actions dependent on a given action, and finding a necessary enabling set for an action), it is necessary to be able to generate the set of *all* actions that add/ remove a given token from a given place (or an arbitrary non-null token, if $a$ has a blocked queue) - and this includes actions that are not currently enabled. If we have the set up shown in Figure 4.3 where $x$ is the

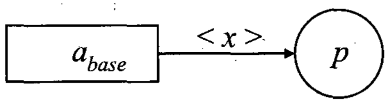


FIGURE 4.3: Finding the set of bindings for $a_{base}$ that add the token $t$ to $p$ is easy in this case.

only variable associated with $a_{base}$, then it is not too difficult to generate all possible bindings for the base action $a_{base}$ that will place a given token $t$ in $p$. However, if the arc inscription expression is more complex, containing conditionals and functions and several variables, then the problem of generating all bindings for $a_{base}$ that add a given token $t$ to $p$ is much, much more difficult. Also, as we will see in the next sub-section, if we wish to find all actions dependent on an action $a$ such that $h(a) \neq \varepsilon$ and $h$ compatibility is required, we must know all of the possible actions $b$ such that $h(b)$ is non-hidden and not equal to $h(a)$. For these reasons (and a few others to come), we suggest that an implementation compute *all* valid actions *before* exploration of the state-space begins, and that a lookup table of which actions add token $t$ to place $p$ for all places $p$ and all $t$ in the domain of $p$.

In order to do this, we suggest restricting the domains of places to a finite number. This is achieved in our formulation by simply not providing the means to declare place domains as infinite sets. It is then relatively easy to use the finiteness of each place domain in order to generate all possible bindings for each base action (disregarding invalid actions), and hence build the lookup table.

This approach has a number of other benefits: for example, we may evaluate the arc inscriptions of the actions and build a look up for these, eliminating the need to evaluate them every time we are deciding whether a base-action may fire with a given binding, and what tokens are added/ removed from the input/ output places of the base action if it

[10]An action $a$ may have several deficient place and token pairs, and so there may be several possible necessary enabling sets for $a$ at $s$. We are not obligated, however, to pick more than one, although as mentioned earlier, the size of the resultant persistent set can vary widely based on our choices of necessary enabling sets.

does. Also, the finiteness of the set of possible tokens that can occur in the net allows us to enumerate the possible tokens, thus imposing a natural, rapidly evaluated ordering on the set of tokens - since several storage schemes for the set of states encountered during the state-space exploration rely on an ordering on the set of states (e.g. balanced trees), which in turn usually requires an ordering on the set of tokens, this can be quite a boon in terms of computational efficiency. Plus, of course, it enables us to find all non-hidden actions in advance.

Computing small stubborn sets in minimal time is no easy task; while the basic algorithm is relatively simple, we see that there are two points where we are asked to make non-deterministic choices: Firstly, at Line 2 where we pick the first enabled action that will add act as a "seed" for the stubborn set; and secondly, at Line 8 where we are asked to find *NES*$(a, s)$ to prevent $a$ from firing, which requires us to choose a deficient place and token pair for $a$ at $s$, and these choices can dramatically affect the size of the final stubborn set (and hence, the final persistent set). Since we wish the set to be as small as possible (as small persistent sets tend to give greater reduction in the size of the resulting state-space, and constructing a large stubborn set gives a greater computational overhead), it is beneficial to have some set of heuristics that can be efficiently employed to prevent the stubborn set from growing out of control. One such heuristic that immediately springs to mind is that the initial action should have as few dependent actions as possible; this helps prevent an initial "surge" of actions being added to the set.

Firstly, then, we note that weak-independence admits the possibility of what may be called "always-persistent actions". These are actions that are weakly-independent of *all* other actions in the system, and so will always form a persistent set if taken on there own. That is, if one of these "always-persistent" actions, $a$, say, is enabled at a state $s$, then the set $\{a\}$ is a perfectly valid (and indeed, optimal) persistent set at $s$. Always persistent actions may be immediately spotted from the static topology of the Petri net: if a base-action $a$ does not share an input place with any other base-action, then for any binding of its variables, $a$ plus the binding is an always persistent set[11]. We can exploit this in the stubborn set algorithm by including a first line that simply checks whether any "always-persistent" actions are enabled at $s$; if we find one, then we may terminate the algorithm immediately and return a persistent set consisting of just this one action [the diagram in Figure 4.4 gives a clue as to which actions to target in this initial search]. This implies that as well as returning smaller sets, the new result could well significantly decrease the computational overhead involved in computing these sets.

All of this is useless, of course, unless these always-persistent actions are reasonably ubiquitous in real life situations. Thankfully, this does indeed seem to be the case; for example, in the example low-level Petri net in Figure 3.1, the actions (*rej2* and *req2*)

[11]If some kind of $h$-compatibility is required in the weak independence relation, for some abstracting homomorphism $h$, then we also require of course that these actions are hidden by $h$.

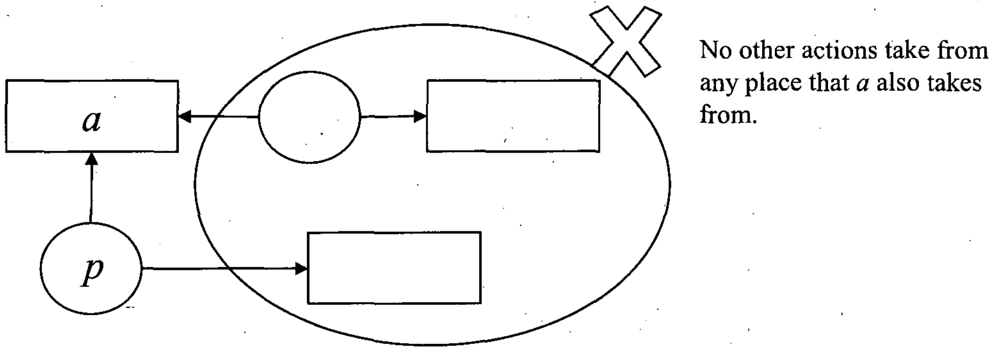

a
p
No other actions take from any place that a also takes from.

FIGURE 4.4: Topological characteristics of an always-persistent action.

are always-persistent, with the actions (*rej1* and *req1*) only prevented from being such by the restriction of $h$-compatibility with the abstracting homomorphism $h$ used in that example.

A concept that is probably more useful (in that it will occur more commonly in real-life examples) is found in "always-persistent" pairs. These are actions that depend *only* upon each other and so, if they are both enabled at a state, will form a persistent set when taken as a pair. Thus, if we have found no always-persistent actions enabled at the current state, we may begin a search for always-persistent pairs. Again, a glance at the topology of the Petri net gives a clue as to which base-actions we should start searching first.

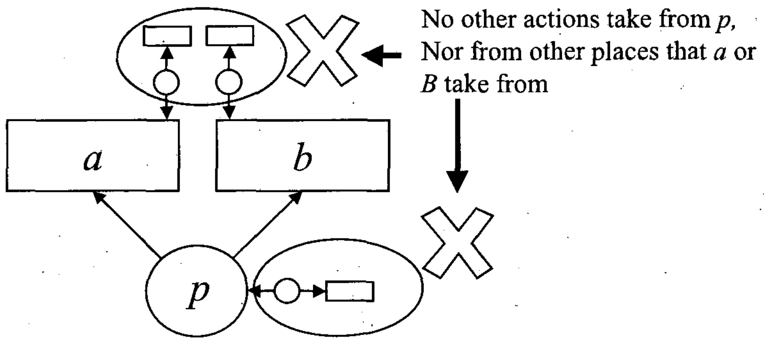

No other actions take from p, Nor from other places that a or B take from
a
b
p

FIGURE 4.5: Topological characteristics of an always-persistent pair.

Let $a^{v_a}$ and $b^{v_b}$ be two actions that have $a$ and $b$ as their base-actions, respectively: then if the token taken by the actions $a^{v_a}$ and $b^{v_b}$ is the same, then these two actions are dependent on each other and moreover will quite probably occur "in tandem" throughout the state-space i.e. $a^{v_a}$ will be enabled at a given state if and only if $b^{v_b}$ is[12]; this enables us to rapidly construct a persistent set consisting of just these two actions. This of course presupposes that no other bindings of variables to the base actions remove the required token from the input place $p$; usually, though, this is not a problem as the

[12] This is because whether an action can fire at a state usually depends more on whether the required tokens from the input places are available at that state than on any other "guard" conditions. There are, of course, exceptions to this, so we must exercise some care.

functions determining which tokens are taken are usually injective mappings in the associated variables. If this is not the case, and these other actions are enabled at the current state, then we can include these actions in the set and hence generalize to always-persistent *triples*, or *quadruples*; etc.

At the other extreme, actions that add non-null tokens to queues should be strongly discouraged from being chosen as our initial action (or indeed, as the action added to the stubborn set as it is required to enable an action $a$, as in Line 8) as, according to Theorem 4.19, *every* other action that adds a non-null token to the same queue will need to be added to the set, and this set of actions is often quite large.

On the subject of queues, queues offer some interesting strategies that may aid in the efficient construction of small persistent sets, or even in reducing the size of the state-space by more direct means. We deal with the latter, more interesting assertion first. Consider a queue $q$ with a capacity of *maxinqueue* whose elements belong to a domain $d$. At any state $s$, the contents of $q$ could have any of

$$1 + |d| + |d|^2 + ... + |d|^{maxinqueue}$$

(which is $O(|d|^{maxinqueue})$) different values. In fairly asynchronous systems, there is more of a tendency for the contents of the places to vary more-or-less independently over the range of their allowable contents, implying that a (very rough) approximation to the size of the state-space can be given my multiplying the sizes of all possible contents of the each place together (assuming these sizes are finite, of course[13]). All of this very vague reasoning is meant to illustrate the following (hopefully non-contentious) point; if we can restrict the set of values that the contents of a place ranges over during the exploration of a state-space, we may directly reduce the size of the resulting state-space.

With queues, it seems that we quite often *can* do this, by adding a bias to the stubborn set construction algorithm towards including actions that remove tokens from queues as the initial choice of a seed for the stubborn set. In principle, this will work well if including such an action does not necessarily imply that an action that adds another token to the queue must always be included in the final persistent set[14]; in the absolute ideal case, every time the queue has a token added to it, we can remove it straight away, ensuring that the queue never contains more than one element at a time, thus giving the desired restriction on the set of values the contents of the queue may take, and in turn

[13]A multi-set or unbounded queue over a finite domain $d$ can technically take an infinite number of values; in practice, however, the "multi" aspect of multi-sets tends to be seldom used (so the set of contents of a place at a state $s$ tends to be a straight *set* drawn from the place domain), and the structure of the system helps to add a bound to the queue (or else the system would be infinite).

[14]Our analysis of dependence shows that an action that removes from a queue and an action that adds to the queue need not be dependent, so, barring any other sources of dependency between two such actions (e.g. perhaps they are not $h$-compatible, or perhaps they take the same token from the same place) this is not at all unlikely.

reducing the size of the state space. Of course, this ideal state of affairs probably will not be attained in practice, but we believe that we may reasonably expect a high-degree of reduction.

The former benefit of queues mentioned earlier also concerns actions that remove from queues. Apart from making a good "seed" action for stubborn sets, these actions are also a very good choice for the "necessary enabler" actions that are required for the stubborn set algorithm (Line 8) as, typically, the construction of a stubborn set at a state $s$ proceeds as follows: An initial action is chosen, and this action will likely be dependent on other actions. For each dependent action $a$ that is not currently enabled, we must take steps to avoid $a$ being enabled from $s$ "outside" of the final persistent set at $s$, and we do this by choosing a "necessary enabler" action, $b$ say, that is necessary to enable $a$, and add it to the stubborn set. Typically, $b$ has a different base-action to $a$. If $b$ is not currently enabled, then we need to find an action $c$ that is required to enable $b$ from $s$, and $c$ quite often has a different base-action again. We see then that, very loosely speaking, the stubborn set often "grows" around the initial action like an expanding wavefront, encompassing more and more of the net topology as it goes on. Typically, the more of the base-actions from the net topology are required, the larger the resulting persistent set (and the longer the time spent constructing it).

If, however, we are required to find a necessary enabling set for an action $a_q$ that takes a non-null token $t$ from a queue $q$, and that $q$ is non-empty and is currently a deficient place for $a_q$ at state $s$, then we may stop there[15]: the fact that the queue is non-empty but deficient for $a_q$ means that the head of the queue is not the token $t$ that $a_q$ requires from $q$, and that we cannot reach a state where $t$ is the head of $q$ *without the actions that are enabled at s and take a non-null token from q firing first.* Thus, the set of actions (call it $A$) that are enabled at $s$ and take a non-null token from $q$ are a necessary enabling set for $q$. Often, the only actions that are dependent on such actions are other actions that take from the same $q$, in which case $A$ forms a necessary enabling set for *these* new actions, too. So in our conception of the stubborn set as a growing wavefront emanating from the initial action to encompass all of the Petri net, actions that take from queues often act as terminators for this process. Also, this rule is relatively simple to implement in a stubborn set construction algorithm.

The stubborn set algorithm we ended up using in the implementation in Chapter 6 was primitive, and basically involved assigning all actions a "weighting", based on how immediately promising it looked. When looking for a seed action at a state $s$, we would choose the action with the highest weighting; likewise, when looking for a deficient place and token pair for an action $a$ at state $s$, we would pick the one that allowed us to use the $NES(a, s)$ that gave the highest weighting over the sum of actions in the

[15]By which we mean that we cannot necessarily just stop the stubborn set construction algorithm, but we can stop the current "chain" of finding necessary enabling sets, then finding necessary enabling sets for these necessary enabling sets etc.

necessary enabling set. The algorithm was thus "greedy", and may not have given optimal persistent sets, but we must be careful to weigh the cost of computing the persistent set (which will increase if we adopt a "trial-and-error" approach, evaluating many possible seed actions for each persistent set and many deficient place and token pairs for each action we need to disable) against the reduction in total time attained by reducing the size of the state-space explored.

The weighting for a given action was calculated as follows:

1. *Action disables no other action*: Add 5 points.
2. *Action removes a token from a queue*: Add 10 points
3. *Action adds a token to a queue*: Subtract 10 points
4. *Action shares a (non-queue) input place with another action*: Subtract 3 points for each such action.

The weightings are intended to gain from the queue-emptying strategy foremost via Rule 1), while favouring always-persistent actions if no queue-emptying actions are enabled via a combination of Rule 2) and Rule 4). If no such actions are present, we try to look for one that is least likely to have many dependent actions via Rule 4) and, since all actions that add to the same queue are mutually dependent, discriminating against such actions via Rule 3). In practice, this approach seemed to yield reasonably good reductions; practical results are given in Chapter 6.

#### 4.3.4.2 With Abstraction Compatibility

Having detailed some techniques for constructing stubborn sets, we now describe the effects that the requirement of $h$-compatibility has on them. None of the practical examples considered in this thesis involve mapping different actions to the same non-hidden action, so for simplicity, we will assume a "worst case" scenario where, if an action is not hidden by $h$, then it cannot be $h$-compatibly independent of any other non-hidden action.

Without the restriction of $h$-compatibility, the actions dependent on an action $a$ invariably share a place with $a$, and so are fairly easily identified and very localised - an action located deep within a component of a system is likely to be independent of an action deep within a different component. As a consequence, stubborn sets tend to take the form of a set of actions with the seed action as a nucleus, expanding to encompass neighbouring actions of actions currently within the set until no more need to be added. It is quite common for a stubborn set seeded around an internal action of a component to consist solely of actions from that component, which gives a great boost to the state-space reduction resulting from a persistent set selective search due to the fact that by following actions only from one component at each state, we are greatly reducing the state-space explosion which would otherwise occur due to having to consider all interleavings of all components. The concepts of "components" and "local states" will be formalised in the next Chapter, but for now we will present an illustrative example. Consider the variant of the low-level Petri net specified system of Figure 3.1, shown in Figure 4.6.

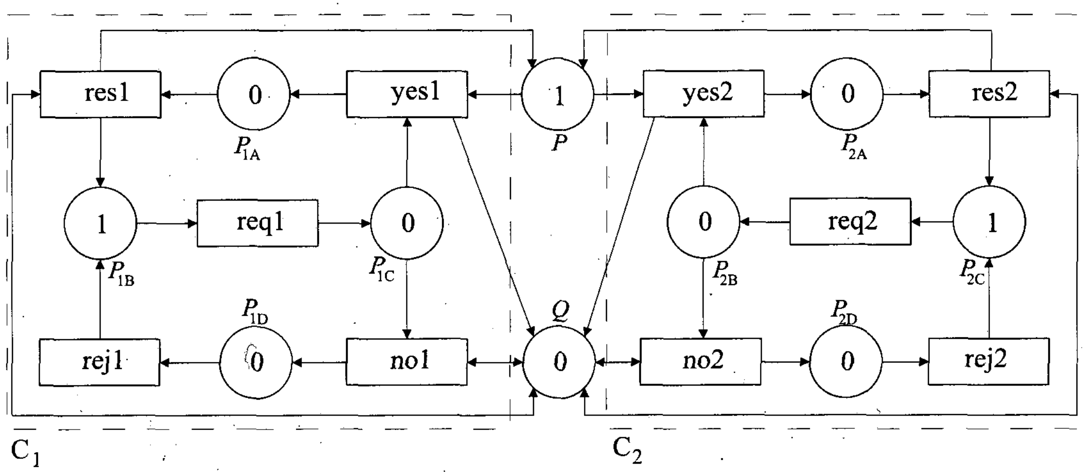


FIGURE 4.6: The system of Figure3.1 decomposed into three components.

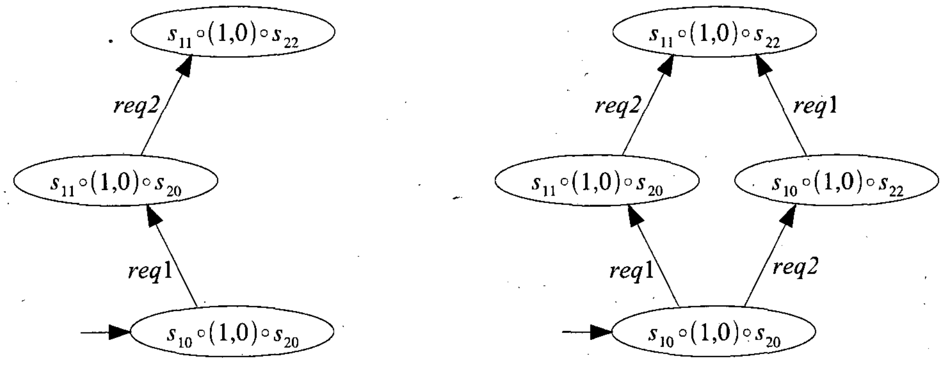
Here, we have decomposed the system into three components: component $C_1$, which contains the actions $res1$, $yes1$, $no1$, $rej1$, $req1$ and the places $P_{1A}$,$P_{1B}$,$P_{1C}$ and $P_{1D}$; $C_2$ which contains the actions $res2$, $yes2$, $no2$, $rej2$, $req2$ and the places $P_{2A}$,$P_{2B}$,$P_{2C}$ and $P_{2D}$; and a central, unnamed component that contains the places $P$ and $Q$. Given a state $s$, the *$C_1$-local state of $s$* is simply the 4-tuple consisting of the contents of the places $P_{1A}$,$P_{1B}$,$P_{1C}$ and $P_{1D}$ in that order (so e.g. the $C_1$-local state of the initial state $s_0$ of Figure 4.6 is $(0,1,0,0)$), and similarly for the $C_2$-local state (so the $C_2$-local state of $s_0$ is $(0,0,1,0)$). The state $s$, then, can be expressed as the combination of the $C_1$-local state of $s$, the contents of $P$ and $Q$, and the $C_2$-local state of $s$ e.g. if we use the symbol $\circ$ to represent composing an $n$-tuple with an $m$-tuple to get an $n+m$-tuple, then we can say

$$s_0 = (0,1,0,0) \circ (1,0) \circ (0,0,1,0)$$

Let $s_{10}$ and $s_{20}$ represent the $C_1$-local and $C_2$-local states of $s_0$ respectively, and let $s_{11}$ represent the $C_1$-local state of the state reached by following $req1$ from $s_0$ and $s_{22}$ the $C_2$-local state of the state reached by following $req2$ from $s_0$. Note that following $req1$ from a state has no effect on the $C_2$-local state and similarly, following $req2$ doesn't affect the $C_1$-local state. If we ignore $h$-compatibility for the time being, then both

$\{req1\}$ and $\{req2\}$ are persistent sets at $s_0$ (in fact, $req1$ and $req2$ are *always-persistent* actions, as they are weakly-independent of all other actions), and are also persistent sets where all actions belong to the same component ($C_1$ in the case of $\{req1\}$; $C_2$ in the case of $\{req2\}$). So, using the definitions just presented, the beginnings of a persistent-set selective search of the state-space of the system of Figure 4.6 (we're ignoring all actions but $req1$ and $req2$, here, and only go exploring a handful of states) could look as shown in Figure 4.7a).

If, however, we introduce an abstracting homomorphism $h$ that is the identity on the actions $req1$ and $req2$ but hides all other actions, then $req1$ and $req2$ are no longer ($h$-compatibly) independent of one another, and so the sets $\{req1\}$ and $\{req2\}$ are no longer persistent at $s_0$, so the only persistent set available at $s_0$ is the trivial set $\{req1, req2\}$; a small portion of the $h$-compatible persistent set selective search is shown in Figure 4.7b). Note that without $h$-compatibility, we avoided having to consider the interleaving of the $C_1$-local state $s_{11}$ with the $C_2$-local state $s_{22}$ but now, since the requirement of $h$-compatibility has introduced a dependency between two actions in the two different components $C_1$ and $C_2$, we are forced to consider this interleaving. In this particular (simple) example, this only introduces one extra state but one can easily imagine the consequences when dealing with a more complex system with more complex local states.


$s_{11}\circ(1,0)\circ s_{22}$
req2
$s_{11}\circ(1,0)\circ s_{20}$
req1
$s_{10}\circ(1,0)\circ s_{20}$
$s_{11}\circ(1,0)\circ s_{22}$
req2
req1
$s_{11}\circ(1,0)\circ s_{20}$
$s_{10}\circ(1,0)\circ s_{22}$
req1
req2
$s_{10}\circ(1,0)\circ s_{20}$

(a) Without $h$-compatibility (b) With $h$-compatibility

FIGURE 4.7: Part of a persistent set selective search of the system shown in Figure 4.6

Thus, $h$-compatibility is problematic not only because it can increase the amount of actions that are no longer independent of a non-hidden action (which can lead to larger persistent sets, and thus less of a reduction in state-space size), but also because the actions that are added as dependent can be located in entirely different parts of the system, potentially weakening the reduction we previously obtained by not having to consider all interleavings of local states.

When experimenting with the stubborn set creation algorithm with our Sliding Windows example and the chosen abstracting homomorphism $h$, we found that the weighting algorithm dealt very badly with the added restriction of $h$-compatibility: in fact, we had to bias against choosing hidden actions to a degree that was very surprising to us. With hindsight, the reason for this is obvious: the reduction in the weak-independence relation incurred by $h$-compatibility would already be troublesome if all newly added dependencies were only between actions that are in the same component, but when it involves creating dependencies between actions that are in *different* components, its effects on the reduction obtained by partial-order reduction are much more damaging for the reasons just described. The example in Chapter 6 had this quality: it could loosely be decomposed into a "sending" component and a "receiving" component, with many of the actions in one component being independent of many of those in the other, but the particular abstracting homomorphism introduced new dependencies between actions in different components, with very damaging effects. Fortunately, however, when a sufficiently large weighting penalty for non-hidden actions was incorporated, the requirement of $h$-compatibility made very little difference to the reduction obtained; again, fuller details are provided in Chapter 6. The final rule was as follows:

5. *Action is not hidden by h*: Subtract 13 points

Another problem with $h$-compatibility, briefly alluded to earlier, is the problem of finding all actions that are dependent on a given action $a$, a necessity for the stubborn set technique to work: when $h$-compatibility is ignored, we (informally) need only consider actions that share a place with $a$, whereas this is not the case when we deal with $h$-compatibility. Our solution was simply to ensure that all places have finite domains so that all possible actions of the system can be known before we begin an exploration: then, the problem of finding all actions $b$ such that $h(ab) \neq h(ba)$ becomes trivial. If this is not an acceptable solution for a given situation, however, then we would need to find an alternate solution.

## 4.4 Summary

We have chosen to use an implementation of the Sliding Windows Protocol to gauge the practical usefulness of the results of Chapter 3 and, since specifying this using low-level Petri nets would be very difficult and time-consuming, we have created and presented a dialect of Petri nets more suited to specifying the protocol, which features native "queue" places. One of the pre-requisites for using the result of Chapter 3 to decide properties of a system is a means of constructing abstraction-compatible persistent sets that also deal with the newly-introduced queue places, and so we have described a means of constructing these persistent sets, based heavily on Valmari's *stubborn set technique*.

The stubborn set technique involves creating a set of actions (each of which may or may not be enabled at the current state) starting with a seed action and progressively adding actions in order to ensure that no actions that are dependent on any of the actions in the set that are currently enabled can fire, unless one of the enabled actions currently in the set fires first. It can be shown that when no further actions need be added, the intersection of this stubborn set with the set of currently enabled actions is persistent. An argument concerning which actions are liable to be enabled "outside" of a persistent set showed that we need not consider the "full" definition of independence in Definition 2.19 when computing persistent sets: a weaker definition called *weak independence* will suffice, and with this, a pair of actions are deemed dependent when they either lead to a different state when followed in a different order, or when one may disable the other. With this relaxed rule, the result of the stubborn set technique is still a persistent set in the original sense. The stubborn set technique is non-deterministic at several steps, so we adopted a crude "weighting" algorithm to help make reasonably effective decisions at each step.

Incorporating queues into the stubborn set technique is largely trivial: we need only revise the dependence rules for pairs of actions that either add or remove a non-null token from the same queue place, and these rules are simple. Abstraction-compatibility is, however, potentially much harder to add: not only can it reduce the number of pairs of actions that are deemed to be independent, leading to larger persistent sets and less state-space reduction but, more importantly, it can create dependencies between previously independent "components" of the system, and an abstraction-compatible persistent-set selective search where the stubborn set weighting algorithm is left untouched will likely explore more interleavings of local states of system components and dramatically lessen the effectiveness of the state-space reduction, as we found to be the case with our practical experiments with our Sliding Windows Protocol implementation. However, in this case at least, it was found that simply adding a strong bias in the weighting algorithm against non-hidden actions negated this to a large degree.

We now go on to give an overview of compositional verification. We will consider compositional verification only as applied to Petri-net-specified systems.

# Chapter 5

# Compositional Verification

## 5.1 Overview

Systems may often be decomposed into (or built up from) two or more components, which may be viewed as separate processes running more or less independently of one another, communicating via shared resources or specialized communication channels[1]. By attempting to *isolate* components from one another, and explore only the *local state space* of each component separately (in contrast with the full, or *global* state space of the system), it may be possible to greatly reduce the size of the state-space examined due to the fact that we may not need to consider all possible interleavings of each component's local states as we would in a typical state-space exploration.

The situation where we wish to construct the full behaviour of a single component appears especially often in the context of *feature interaction*. Here, new "features" (*components*) are added to the "core" system, and we wish to check whether each feature satisfies certain properties: For example, if our core system was some kind of exchange server, we might want to "plug in" a feature such as call-forwarding and check whether this feature behaves as desired in the context of the core system. As more features are added over time, they are bound to interfere (*interact*) with the existing features, either directly if they share the same resources, or indirectly through the knock-on effect of the changes to the core system behaviour induced by adding the new features. It is useful, then, to be able to construct the behaviour of just one or more of these features in order to verify that it still behaves correctly, preferably without having to construct the whole system behaviour, and this is precisely what this result enables us to do.

Naïve attempts to isolate a component simply by severing it from the rest of the Petri Net, however, results in problems related to the *binding problem* of the previous Chapter,

[1] This 'disintegration' into components is fairly natural with Petri nets, since they are most often constructed component-wise, usually with each component being situated on a different 'page', or at least being "geographically" separate from other components.

where restrictions on the firing of actions arising from the removal of restrictions on the possible values of variables corresponding to the action can possibly lead to local states in the isolated components that have no counterpart in the full system. The appearance of *false local states* such as these and the possible resulting explosion of the local state space exploration of a component are part of what is called the *enviroment problem.* The main result of this chapter is to leverage the abstraction-compatible persistent-set technique of Chapter 3 to solve the environment problem for a specific configuration of components, allowing us to efficiently construct the entire behaviour and set of local states of a given component based on a partial-order reduced version of the full system. Solving the enviroment problem is the sole focus of this chapter: we do not consider other aspects of Compositional Verification, such as decomposition strategies and deduction of properties satisfies by the full system based on properties satisfied by its individual components.

Unlike the result of Chapter 3, this result is explicitly tied to Petri-net specified systems, but not to any particular formulation of Petri nets. We define *abstract Petri nets* (of which the low-level Petri nets of Chapter 2 and the high-level Petri nets of Chapter 4 are merely specific formulations) to simplify and generalize the proofs. After this, we give some more formal definitions of concepts such as local states and go on to describe the specific configuration of systems our result applies to, and describe some existing techniques that attempt to solve the environment problem. We then present the algorithm we used to solve the environment problem and enable the efficient construction of a full component's behaviour, and prove that it works. We then show that the restriction to the specific system configuration we have chosen may in fact not be necessary after all. Finally, we describe a much more straightforward method of constructing the behaviour of a component using abstraction-compatible partial-order reduction, and attempt to justify why the more complex result of this chapter should be used instead.

The result of this chapter was presented at VVEIS'03 [St James and Ultes-Nitsche (2003)].

### 5.1.1 Abstract Petri Nets

In preparation for the statement and proof of the result, we present an alternative formulation of high-level Petri nets (which we will call *abstract Petri nets*) to give a more abstract, generalized formulation. This is done both to simplify the proofs and to avoid restricting the applicability of the result to the specific formulation of Petri nets we presented in Chapter 4. This generalization is intended to preserve key aspects of Petri net definitions (such as the one presented in Chapter 4), namely the use of place contents to specify a state; the fact that the enablement of s base-action $a$ with a given binding at a state is determined by a guard condition along with a condition on the contents of the set of input and output places of $a$; and the fact that the state reached by firing a base-action $a$ with a given binding differs from the preceding state only in the contents of the set of input and output places of $a$. The generalization is inspired by the definition in Chapter 4 (indeed, such Petri nets are merely a specific case of these abstract Petri nets - we will show this by drawing parallels between the definition of Abtract Petri nets and the Petri nets described in Chapter 4 .)

Before this, a small definition that will be useful throughout this section.

**Definition 5.1.** Let $X \subseteq \Sigma_{base}$ be any subset of the set of base-actions. Then we define $X^v$ as the set of all base-actions $a$ in $X$ together with the set of bindings of the variables of $a$. In other words, $X^v$ is the set of all actions whose base-action belongs to $X$.

**Definition 5.2.** An *Abstract Petri Net* $\mathcal{P}$ consists of the following:-

A set $P$ of places. Each place $p_i \in P$ contains a multi-set of elements drawn from some domain, $d_i$.

A set $\Sigma_{base}$ of base-actions. Each base-action $a \in \Sigma_{base}$ has a set $x_1^a, x_2^a, ..., x_{m^a}^a$ of variables, a set $In_a \subseteq P$ of input places, and a set $Out_a \subseteq P$ of output places. It also has two functions associated with it; more on this later.

A *state* of a Petri Net is a mapping from each $p_i \in P$ to a multi-set of $d_i$ (i.e. an assignment of *contents* to each place $p_i \in P$; in low-level Petri nets, this would be the number of tokens in place $p_i$ ). Let $s(p)$ denote the contents of place $p \in P$ at state $s$. Let $S$ be the set of all possible states; i.e. the set of all mappings of the type described.

An initial *state*, $s_0 \in S$.

So far, the definition of an abstract Petri net mirrors the definition of a high-level Petri net given in Definition 4.6 very closely.

Each base-action $a \in \Sigma_{base}$ has associated with it a *guard* predicate,

$$guard_{a,p_i}(s(p_i), x_1^a, x_2^a, ..., x_{m^a}^a)$$

defined for each $p_i \in Out_a \cup In_a$. $s(p_i)$, rather than just $s$, is explicitly used to emphasize that $guard_{a,p_i}$ is dependent solely on the the contents of place $p_i$ in $s$, rather than the contents of all other places as would be the implication if we had used $s$ instead. . If $a \in \Sigma_{base}^v$ (Definition 5.1; i.e. if it is a base-action together with a set of values for each of the variables $x_1^a, x_2^a, ..., x_{m^a}^a$), then we abbreviate the guard predicate as $guard_{a,p_i}(s(p_i))$.

An *action* in an abstract Petri net is defined as in Definition 4.8 - it is a pair consisting of a base action plus a binding of values to the set of variables associated with that base action. From Definition 5.1, an action belongs to the set $\Sigma_{base}^v$.

An action $a \in \Sigma_{base}^v$ is *enabled* at a state $s \in S$ if and only the guard predicate

$$guard_{a,p_i}(s(p_i), x_1^a, x_2^a, ..., x_{m^a}^a)$$

is true for each $p_i \in Out_a \cup In_a$. This is basically an abstraction of the rules concerning enablement of actions at a state, as in Definition 4.10. Instead of dealing with complex, concrete rules dealing with input and output arcs and labels, we have simply abstracted the most essential information piece of information: that whether an action may fire at a given state $s$ is dependent solely on the values of the bindings and on the contents of the input and output places of $a$ at $s$.

The second function associated with $a \in \Sigma_{base}$ is $change_{a,p_i}(s(p_i), x_1^a, x_2^a, ..., x_{m^a}^a)$. This function is defined for each $p_i \in Out_a \cup In_a$ and returns a multi-set drawn from $d_i$. Again, if $a \in \Sigma_{base}^v$, we abbreviate the function by $change_{a,p_i}(s(p_i))$. If $a \in \Sigma^v$ is enabled at $s \in S$ then the state $s' \in S$ reached from $s \in S$ by following the action $a \in \Sigma^v$ is defined as:

$$\begin{array}{ll} s'(p) = s(p) & \forall p \in P - Out_a \cup In_a; \\ s'(p) = change_{t,p}(s(p)) & \forall p \in Out_a \cup In_a \end{array}$$

This final function encapsulates the essential part of Definition 4.11 - that the firing of action $a$ takes us to a new state where the only differences from the old state are the contents of the input and output places of $a$.

We formalise the idea of a "local state" of a component as follows. Note that when dealing with a component and its local states, an ordinary state $s \in S$ in an abstract Petri net $\mathcal{P}$ is often described as a *global state*, solely to differentiate it from a local state.

**Definition 5.3.** Let $\mathcal{P}$ be an abstract Petri net with a set $P = (p_1, p_2, ..., p_n)$ of $n$ places and full set of states $S$, and $X \subseteq P$ be a subset of $m$ of the set of places. Let $s \in S$ be any state. Then the *$X$-local state of* $s$ is the $m$-tuple obtained by taking the $n$-tuple $s$ and simply removing the components that do not represent the contents of a place in $X$. More formally, let $x$ be an injection from the set $\{1, 2, ..., m\}$ into the set $\{1, 2, ..., n\}$ $(m \leq n)$ such that $x(i) < x(j)$ whenever $i < j$ (such $x$ can be placed in a one-to-one correspondence with subsets $X \subseteq P$ by defining $X = \{p_{x(1)}, p_{x(2)}, ...p_{x(m)}\}$), and $s$ be a state in $S$. Then the $X$-local state of $s$ is the $m$-tuple $(s(p_{x(1)}), s(p_{x(2)}), ..., s(p_{x(m)}))$.

If $s'$ is an $X$-local state of any $s \in S$, then $s$ is said to be a *global state corresponding to the local state* $s'$. If there is no such $s$ in the set of states obtained in a state-space exploration of the abstract Petri net $\mathcal{P}$, then $s'$ is called a *false local state*. A *local state* is simply an $|X|$-tuple which is an $X$-local state of a state $s \in S$ for some $X \subseteq P$.

Let $C \subseteq \Sigma_{base}$ be a subset of the set of base-actions in $\mathcal{P}$; then a *$C$-local state* of $\mathcal{P}$ is an $X$-local state of $\mathcal{P}$ with $X = \bigcup_{a \in C} Out_a \cup In_a$ i.e. it is the local state restricted to those places that are an input/ output place of some $a \in C$.

## 5.2 The Problem and Existing Techniques.

The main result of this Chapter is geared towards systems that have a particular decomposition into components. In these systems one component, which we will call $C_1$, is the *only* one whose behaviour we are interested in. Direct communication between this component and the rest of the system, $C_2$, is prohibited, and all such communications must be passed through what we will call the *interface component*, denoted $I$. Usually, $C_1$ and $C_2$ are "larger" than $I$ in terms of numbers of actions; indeed, in such systems, the interface-level often appears as a thin strip of actions sandwiched between two large sub-Petri nets representing $C_1$ and $C_2$ (see Figure 5.1). Since $C_1$ and $C_2$ may not communicate directly, the type of system we will concentrate on looks, schematically, like that in Figure 5.1. The arrows indicate informational flow [so $C_1$ and $C_2$ may "communicate" only via $I$].

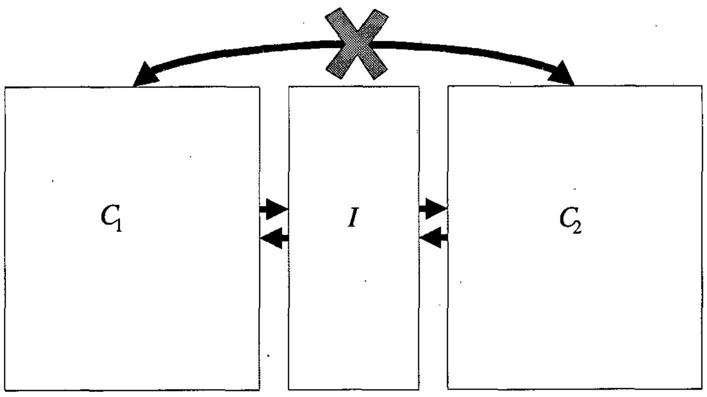


FIGURE 5.1: Schematic of the type of situation we will concentrate on in this section.

As mentioned, it is theoretically possible to derive large reductions in the state-space from considering just the behaviour of the required component in isolation from the others, and we describe now the problem that occurs when attempting to do this. "Isolating" a component in a Petri net is a very convenient operation: one simply forms a new Petri net by removing all actions not in the component one wishes to investigate, and removing all places except for ones accessed *only* by the actions in our desired component, and explores this new net. The "dummy" example of Figure 5.2 shows this [the component $C_2$ is ignored for the sake of simplicity].

Perhaps counter-intuitively, this Petri net contains at least as many behaviours from the point of view of component $C_1$ as the original . This is because the Petri net semantics tells us that removing input and output arcs from an action in a Petri net merely *removes* restrictions on whether that action may fire. To illustrate why this is a problem, consider a flawed system where $C_1$ is a component that takes requests from $C_2$ using $I$ as an intermediary (see Figure 5.3), and which has undefined behaviour if called upon to process a certain pair of requests (say, *req_a* and *req_b*) consecutively. Assume that the system has been designed in such a way that $I$ and $C_2$ ensure that $C_1$ is never

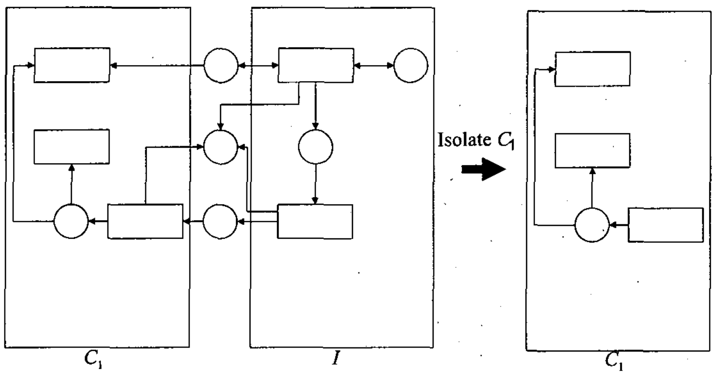


FIGURE 5.2: "Isolating" a component.

handed this pair of requests . Performing the severing step to isolate $C_1$ as shown in Figure 5.3, we see that $C_1$ can choose to receive, at any state, any possible request (the severed $C_1$ is said to operate in a *maximal environment*[Kupferman and Vardi (1997)]), and so can end up trying to process sequences of requests that were forbidden in the full system, including the pair *req_a* and *req_b* that can lead to undefined behaviour. We call such computations that occur in the Petri net formed by isolating $C_1$ and which have no counterpart in the full Petri net *false computations*, and the local states that they can lead to are often false local states . It is not unusual for the lack of restrictions and regulations of data provided to $C_1$ to create a great many such false local states, leading to a phenomenon which might be termed the *local state-space explosion*, where the number of local states explodes due to proliferation of false local states created due to the lack of external regulation incurred when we isolate $C_1$ from the rest of the system.

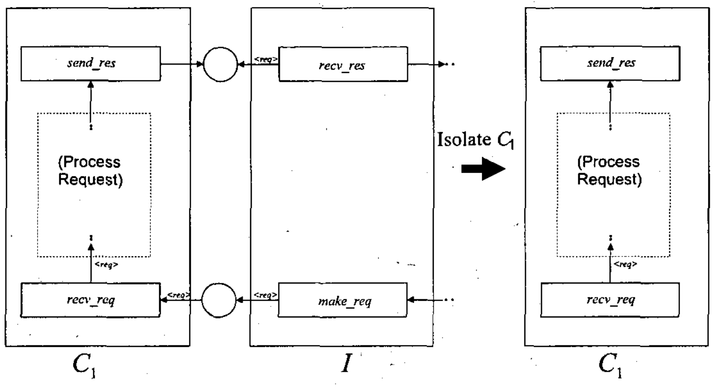


FIGURE 5.3: A problem with isolating components.

We are also potentially faced with a problem with finding the possible bindings for the *req* variable associated with the base-action (see Definition 4.6) *recv_req* which is necessary for finding the set of enabled actions at a state - if the domain of *req* is finite, then it is not too difficult - we must simply assign all possible values in the domain to *req* (which will likely be many more values that could be assigned if *recv_req*'s input place had not been removed - this is the main source of the local state-space explosion). If, however, it is infinite, then we will likely be unable to proceed as we cannot in general know if we have considered all possible bindings to *req* with which *recv_req* may fire, and so cannot generate the full set of enabled actions at the given state.

The problems with isolation defined here - particularly the creation of false computations and the resulting false local states - are called *the environment problem* [McMillan (1997, 2000)]. The environment problem occurs in any situation where we remove the corrective influence of the rest of a system from a component of the system, and is not restricted to Petri nets. There are two main ways of dealing with the environment problem: the *assume-guarantee*[Pnueli (1985); Jones (1983)] technique, and the *compositional minimisation*[de Alfaro and Henzinger (2001); M. Chiodo et al. (1992)] technique.

The *assume-guarantee* technique attempts to solve the problem of the isolated $C_1$ component acting in ways not permissible in the full system by using *assumptions* about how the rest of the system ($I$ plus $C_2$, in our example) behaves when verifying properties about $C_1$ e.g. "$I$ and $C_2$ will never submit requests *req_a* and *req_b* to $C_1$ consecutively". This can reduce some of the aberrant behaviours that would otherwise be seen after $C_1$ has been isolated. Having then verified properties of $C_1$ under the given assumptions, we must then prove that the assumptions do indeed hold (e.g. prove that $I$ and $C_2$ do indeed never submit the problematic pair of requests). The main drawbacks of the assume-guarantee technique are that these initial assumptions about how the rest of the system behaves must often be specified manually by an engineer, usually after several rounds of trial and error verification of the isolated $C_1$ i.e. the verification is performed and, if any behaviours of the isolated $C_1$ are observed that are judged to be impossible in the full system, then assumptions necessary to prevent these behaviours are added to the list of assumptions about the rest of the system, and the verification is attempted again.

Another drawback is the possibility for circular reasoning when verifying that our assumptions about the rest of the system (again, this corresponds to $I$ plus $C_2$ in our example) are indeed correct: verifying the full, composed system negates the point of performing compositional verification so we must make assumptions, when verifying that $I$ plus $C_2$, about the behaviour of $C_1$, which can leads to tricky exercises in reasoning[Barringer and Giannakopoulou (2003)]. Also, we aim to construct the local behaviour of $C_1$ *exactly*, and it is not clear that the level of detail about $I$ and $C_2$ required to achieve this can be captured by a list of assumptions. These drawbacks make the assume-guarantee approach a poor fit for the problem at hand, so we shall expand on the technique no further.

*Compositional minimsation* takes a different approach: here, the rest of the system $I$ plus $C_2$ is replaced by a surrogate component that behaves, from the point of view of $C_1$, *exactly* as $I$ plus $C_2$ does. If this surrogate component is smaller than the original $I$ plus $C_2$ (as is quite likely, as it must model *only* the behaviours which are directly visible to $C_1$, and no others), then great gains in efficiency can be made during the verification of $C_1$. This, of course, leaves open the question of how this surrogate component can be created in the first place (although it appears, in general, to be easier to automate than the assume-guarantee technique[de la Riva and Tuya (2006)]) and it is this question which will explore here. The main result of this chapter can be summarised as a compositional minimisation approach to the environment problem for a wide class of Petri nets that uses partial order reduction and abstracting homomorphisms to create a surrogate component that represents, from the point of view of $C_1$, exactly the behaviour of $I$ plus $C_2$, enabling the full behaviour of the $C_1$ to be constructed with no false computations or false local-states created.

### 5.2.1 Statement and Proof of Result.

In this section, we formalise some key concepts and state and prove our result. We begin by formalising the separation of a net into components $C_1$, $C_2$ and $I$ such that $C_1$ and $C_2$ may not communicate directly. We assume throughout that $\mathcal{P}$ is an instance of an Abstract Petri net with set of base-actions $\Sigma_{base}$ (a Petri net as per the definitions in Chapter 4 will do, as previously noted). Let $L_{\mathcal{P}}$ be the language represented by this automaton.

**Definition 5.4.** Let $\Sigma_{base}$ be the set of base-actions of $\mathcal{P}$. Then a $C_1 - I - C_2$ *componentisation of* $\mathcal{P}$ is a split of $\Sigma_{base}$ into three subsets $c_1$, $c_2$ and $I$ such that:

- $c_1$, $c_2$ and $I$ are mutually disjoint;
- $c_1 \cup c_2 \cup I = \Sigma_{base}$; and
- $\forall a \in c_1, b \in c_2, (Out_a \cup In_a) \cap (out_b \cup In_b) = \phi$.

The last condition is the interpretation of the requirement that $C_1$ and $C_2$ may not communicate directly: we enforce this by stipulating that no action in $c_1$ may share an input or output place with any action in $c_2$ (and vice-versa). Note that since $c_1 \cup c_2 \cup I = \Sigma_{base}$, we have (using the notation of Definition 5.1) that $c_1^v \cup c_2^v \cup I^v = \Sigma_{base}^v$. The key point of the result of this section is that given this restriction on $c_1$ and $c_2$, any action in $c_1^v$ is *independent* of any action in $c_2^v$, so that the firing of any action in $c_2^v$ at a state $s$ cannot affect whether any action in $c_1^v$ may fire at the reached state. As a consequence, whether an action in $c_1^v$ may fire is determined solely by which actions in $c_1^v \cup I^v$ have fired previously. We exploit this fact by constructing just the *interface level behaviour* of $\mathcal{P}$ (defined shortly) and subsequently using this behaviour to decide when a given action $a \in c_1^v$ may fire, without any reference to actions in $c_2^v$. This forms the "surrogate component" mentioned in the description of compositional minimisation. .

**Definition 5.5.** Let $\Sigma$ be an input alphabet, $X \subseteq \Sigma$. Then the mapping

$$h_X : \Sigma \rightarrow \Sigma \cup \{\varepsilon\}$$

is defined by

$$\begin{aligned} h_X(a) &= a \quad \forall a \in X \\ h_X(a) &= \varepsilon \quad \forall a \notin X \end{aligned}$$

This mapping simply preserves actions in $X$ but hides all others.

**Definition 5.6.** Let $\Sigma$ be an input alphabet, $X \subseteq \Sigma$, $h_X$ be the mapping as defined in Definition 5.5. Then we denote the minimal automaton representing the language $h_X(L_{\mathcal{P}})$ by $\mathcal{A}_X$. We write $L_X$ as an abbreviation for $h_X(L_{\mathcal{P}})$, the language accepted by $\mathcal{A}_X$.

**Definition 5.7.** The *interface-level behaviour* of a Petri net $\mathcal{P}$ representing language $L_{\mathcal{P}}$ with a $C_1 - I - C_2$ componentisation of its base-actions is the language $h_{I^v}(L_{\mathcal{P}})$ i.e. the language obtained after hiding all actions in words of $L_{\mathcal{P}}$ that are not in $I^v$.

We employ the partial-order reduction technique [St James and Ultes-Nitsche (2001) and Chapter 3 of this thesis] (or perhaps the successor to this result, presented in Appendix B) in order to find the automaton $\mathcal{A}_{I^v}$- from Definitions 5.6 and 5.7, this represents the behaviour of the component $I$ . We then form a new Petri net by "severing" $C_2$ from $I$ and $C_1$ by erasing all actions in $C_2$ and all places connected to actions in $C_2$ and all arcs connected with these places (i.e. the operation is very similar to that performed in Figure 5.2)[2]. We then use the earlier mentioned fact that the enablement of actions in $c_1^v$ at a state $s$ depend only on which actions in $c_1^v$ and $I^v$ have previously fired in order to construct an automaton $\mathcal{A}^{\overline{I^v c_1}}$ that we will eventually prove represents the language $L_{c_1^v \cup I^v}$, from which it is a simple matter to abstract the full behaviour of the $C_1$ component of $\mathcal{P}$, without having to construct the full behaviour of $\mathcal{P}$ first.

**Definition 5.8.** Let $\mathcal{A}$ be any automaton, with transitions labelled by elements from some alphabet $\Sigma$. If the word $w \in \Sigma^*$ may be followed from some state $s$ of $\mathcal{A}$, reaching the state $s'$ of $\mathcal{A}$, then we write:

$$succ(s, w) = s'.$$

**Definition 5.9.** We set $P^{\neg c_2} = P - \bigcup_{a \in c_2} (Out_a \cup In_a)$ (so $P - P^{\neg c_2} = \bigcup_{a \in c_2} (Out_a \cup In_a)$). $P^{\neg c_2}$ corresponds to the set of places that cannot be directly affected by the firing of any action in $c_2^v$ .

[2]Note that many base-actions in $I$ will now have "missing" input places, and so it seems that we run the risk of creating false computations and states. We stipulate that the automaton representing the behaviour of $I$ must contain information about variable bindings precisely in order to circumvent the "binding problem" described earlier, and will show this information is sufficient to ensure that no false computations (and so no false states) are created.

**Definition 5.10.** Construction of $\mathcal{A}^{\overline{I^v c_1}}$ is as follows. Each state in the automaton will be a pair $(s, \overline{s})$, with $s \in S$ and $\overline{s} \in \mathcal{A}_{I^v}$. The states $s \in S$ are a "restricted" Petri Net state in the sense that we only care about the contents of a subset $P^{\neg c_2} \subseteq P$ of the places; the contents of the places $P - P^{\neg c_2} \subseteq P$ may be assumed to be always empty, or set to some arbitrary value; they will play no part in the construction of $\mathcal{A}^{\overline{I^v c_1}}$. Also, the base-actions in $c_2$ are completely ignored during construction.

The initial state of $\mathcal{A}^{\overline{I^v c_1}}$ is $(s_0, \overline{s}_0)$, where $s_0, \overline{s}_0$ are the initial states of $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{A}_{I^v}$, respectively.

The construction proceeds as follows. The action $a \in \Sigma$ may fire at the state $(s, \overline{s})$ if and only if both of the following conditions hold:-

C1) $guard_{a,p_i}(s(p_i))$ is true for each $p_i \in (Out_a \cup In_a) \cap P^{\neg c_2}$; and

C2) if $a \in I_v$, then $a$ is enabled at $\overline{s}$ in $\mathcal{A}_{I^v}$.

The state $(s', \overline{s}')$ reached by following this transition (if enabled) from $(s, \overline{s})$ is defined by:-

$$\begin{array}{ll} s'(p) = change_{a,p}(s(p)) & \text{if } p \in (Out_a \cup In_a) \cap P^{\neg c_2} \\ s'(p) = s(p) & \text{otherwise} \end{array}$$

and

$$\begin{array}{ll} \overline{s'} = \overline{s} & \text{if } a \notin I^v \\ \overline{s'} = succ(\overline{s}, a) & \text{otherwise} \end{array}$$

Two states $(s, \overline{s})$ and $(s', \overline{s}')$ in $\mathcal{A}^{\overline{I^v c_1}}$ are equivalent if and only if $s \sim_{P^{\neg c_2}} s'$ and $\overline{s} = \overline{s}'$.

Let the language represented by this automaton be $L^{\overline{I^v c_1}}$.

Having now defined our terms and the construction of the automaton $\mathcal{A}^{\overline{I^v c_1}}$, we now begin to prove that even though the construction process of $\mathcal{A}^{\overline{I^v c_1}}$ involves removing arcs and places from actions in $I$ that could lead to false states and computations as described earlier, the language $L^{\overline{I^v c_1}}$ represented by $\mathcal{A}^{\overline{I^v c_1}}$ is in fact equal to the desired behaviour $L_{c_1^v \cup I^v}$ (and no such false states and computations are created).

**Definition 5.11.** Let $s, s' \in S$, and let $X \subseteq P$. Then we say that

$$s \sim_X s' \text{ if and only if } s(p) = s'(p) \forall p \in X$$

Such a pair $s$, $s'$ are said to be *X-equivalent* .

Intuitively, two states are $X$-equivalent if and only if the contents of each place in $X$ for each state are equal - the contents of places not in $X$ are ignored in the comparison.

**Lemma 5.12** (if two states are "$X$-equivalent" (for some $X \subseteq P$), then they are still $X$-equivalent after they both follow the same action). *Let $X \subseteq P$ be any subset of the* *set of places, $s$ and $s'$ be two states in $S$ such that $s \sim_X s'$ (Definition 5.11). Let $a \in \Sigma$ be any action that is enabled at both $s$ and $s'$. Then $succ(s,a) \sim_X succ(s',a)$.*

*Proof.* If suffices to show that $succ(s,a)(p) = succ(s',a)(p) \forall p \in X$. Let $p \in X$; there are two cases:

Case i) $p \in Out_a \cup In_a$.

Then $succ(s,a)(p) = change_{a,p}(s(p)) = change_{a,p}(s'(p)) = succ(s',a)(p)$.

Case ii) $p \notin Out_a \cup In_a$

Then $succ(s,a)(p) = s(p) = s'(p) = succ(s',a)(p)$. □

**Lemma 5.13** (actions in $c_1^v$ do not affect local states of $C_2$). *Let $s \in S$. Then if $a \in c_1^v$ is enabled at $s$, leading to $s'$, then $s \sim_{P-P^{\neg c_2}} s'$*

*Proof.* $P - P^{\neg c_2} = \cup_{t \in c_2} (Out_t \cup In_t)$. By assumption,

$$\forall a \in c_1, b \in c_2, (Out_a \cup In_a) \cap (Out_b \cup In_b) = \phi$$

By rules of construction, $s'$ differs from $s$ only in the places in $(Out_a \cup In_a)$. The previous two equations taken together imply that $(Out_a \cup In_a) \subseteq P^{\neg c_2}$. Thus $s'$ differs from $s$ in no places in $P - P^{\neg c_2}$; i.e. $s \sim_{P-P^{\neg c_2}} s'$. □

**Corollary 5.14** (words in $(c_1^v)^*$ do not affect local states of $C_2$.). *Let $s \in S$. Then if $a \in (c_1^v)^*$ can be followed from $s$, leading to $s'$, then $s \sim_{P-P^{\neg c_2}} s'$*

*Proof.* Induction on $|w|$, using Lemma 5.13. □

**Lemma 5.15.** *Let $s \in S$. Then if $b \in c_2^v$ is enabled at $s$, leading to $s'$, then $s \sim_{P^{\neg c_2}} s'$*

*Proof.* Mirrors proof of Lemma 5.13. □

**Corollary 5.16.** *Let $s \in S$. Then if $b \in (c_2^v)^*$ can be followed from $s$, leading to $s'$, then $s \sim_{P^{\neg c_2}} s'$.*

*Proof.* Mirrors proof of Corollary 5.14. □

**Lemma 5.17** (firing of $C_1$ actions is not dependent on the local state of $C_2$). *If $s \sim_{P^{\neg c_2}} s'$ then $a \in c_1^v$ is enabled at $s \Leftrightarrow a \in c_1^v$ is enabled at $s'$.*

*Proof.* In a similar manner to the proof of Lemma 5.12, we deduce that

$Out_a \cup In_a \subseteq P^{\neg c_2}$.Thus, $s(p_i) = s'(p_i) \forall p_i \in Out_a \cup In_a$ [since $s \sim_{P^{\neg c_2}} s'$].

Therefore, $guard_{a,p_i}(s(p_i)) = guard_{a,p_i}(s'(p_i)) \forall p_i \in Out_a \cup In_a \subseteq P^{\neg c_2}$, since $s \sim_{P^{\neg c_2}} s'$. Thus, $a \in c_1^v$ is enabled at $s \Leftrightarrow a \in c_1^v$ is enabled at $s'$; hence result. □

**Corollary 5.18.** *If $s \sim_{P^{\neg c_2}} s'$ then*

*i) $w \in (c_1^v)^*$ may be followed from $s \Leftrightarrow w \in (c_1^v)^*$ may be followed from $s'$; and*

*ii) $succ(s, w) \sim_{P^{\neg c_2}} succ(s', w)$*

*Proof.* Induction on length of $|w|$, coupled with results of Lemmas 5.12 & 5.17. □

**Lemma 5.19.** *If $s \sim_{P-P^{\neg c_2}} s'$ then $b \in c_2^v$ is enabled at $s \Leftrightarrow b \in c_2^v$ is enabled at $s'$.*

*Proof.* Mirrors that of Lemma 5.17. □

**Corollary 5.20.** *If $s \sim_{p-P^{\neg c_2}} s'$ then*

*i) $w \in (c_2^v)^*$ may be followed from $s \Leftrightarrow w \in (c_2^v)^*$ may be followed from $s'$; and*

*ii) $succ(s, w) \sim_{P-P^{\neg c_2}} succ(s', w)$*

*Proof.* Mirrors that of Corollary 5.18. □

**Lemma 5.21** (actions in $c_1^v$ and $c_2^v$ are *independent*). *Let $wabw' \in L_{\mathcal{P}}$ be any action in $L_{\mathcal{P}}$ such that $a \in c_1^v$ and $b \in c_2^v$. Then*

*i) $wbaw' \in L_{\mathcal{P}}$;*

*ii) $succ(s_0, wabw') = succ(s_0, wbaw')$.*

*Proof.* We prove i) & ii) together.

Let $s$ be the state reached by following $w$. Now, $succ(s, a) \sim_{P-P^{\neg c_2}} s$ by Lemma 5.12. Since $b$ is enabled at $succ(s, a)$ [because $wab$ can be read from $s_0$], $b$ is enabled at $s$ [by Lemma 5.17]. Thus, $wb \in L_{\mathcal{P}}$.

Now, $succ(s, b) \sim_{P-P^{\neg c_2}} s$ by Lemma 5.15. Thus, $a$ is enabled at $succ(s, b) = succ(s_0, wb)$. So $wba \in L_{\mathcal{P}}$. Now

$$succ(s,b) \sim_{P-P^{\neg c_2}} succ(s,ba) \text{ by Lemma 5.13} \quad (5.1)$$

$$succ(s,a) \sim_{P-P^{\neg c_2}} s \text{ by Lemma 5.13} \quad (5.2)$$

$$succ(s,ab) \sim_{P-P^{\neg c_2}} succ(s,b) \text{ by Lemma 5.12 and 5.2} \quad (5.3)$$

Eliminating $succ(s,b)$ from (5.3) and (5.1) gives that

$$succ(s,ab) \sim_{P\_P^{\neg c_2}} succ(s,ba) \tag{5.4}$$

Now,

$$\begin{aligned} succ(s,a) &\sim_{P^{\neg c_2}} succ(s,ab) \text{ by Lemma 5.15} && (5.5)\\ succ(s,b) &\sim_{P^{\neg c_2}} s \text{ by Lemma 5.15} && (5.6)\\ succ(s,ba) &\sim_{P^{\neg c_2}} succ(s,a) \text{ by Lemmas 5.12 and 5.2} && (5.7) \end{aligned}$$

Eliminating $succ(s,a)$ from (5.7) and (5.5) gives that

$$succ(s,ab) \sim_{P^{\neg c_2}} succ(s,ba) \tag{5.8}$$

Combining (5.4) and (5.8) gives that

$$succ(s,ab) = succ(s,ba)$$

Thus, $w'$ is readable from $succ(s,ab)$ if and only if $w'$ is readable from $succ(s,ba)$ and moreover, we reach the same state either way. Hence i) and ii) hold. □

**Lemma 5.22** (separating words into $C_1$ and $C_2$ "segments".). *If $ww'w'' \in L_{\mathcal{P}}$, where $w' \in (c_1^v \cup c_2^v)^*$, then*

*i) there is a word $wxyw'' \in L_{\mathcal{P}}$ where $x \in (c_1^v)^*$ and $y \in (c_2^v)^*$; and*

*ii) $succ(s_0, ww'w'') = succ(s_0, wxyw'')$.*

*Proof.* We may transform $ww'w''$ into $wxyw''$ by repeatedly swapping pairs $x_i \in c_1^v$ and $y_i \in c_2^v$ in $w'$ whenever $y_i$ appears directly before $x_i$. The results i) & ii) hold at each step by Lemma 5.19. □

**Lemma 5.23** (a computation in the full system, with $C_2$ actions hidden, is represented in our construction; i.e. "faithfulness" of the construction w.r.t $C_1$ holds in one direction.). *If $w \in L_{\mathcal{P}}$, then*

*i) $h_{c_1^v \cup I^v}(w) \in L^{\overline{I^v c_1}}$.*

*Furthermore, let $s \in S$ be the state of $\mathcal{A}_{\mathcal{P}}$ reached by following $w$, $(s', \overline{s}')$ the state of $\mathcal{A}^{\overline{I^v c_1}}$ reached by following $h_{c_1^v \cup I^v}(w)$. Then*

*ii) $s \sim_{P^{\neg c_2}} s'$ and*

*iii) $\overline{s}'$ is the state of $\mathcal{A}_{I^v}$ reached by following $h_{I^v}(w) \in L^{\overline{I^v c_1}}$.*

*Proof.* First, a little sub-lemma. Parts ii) and iii) hold for $w \in L_{\mathcal{P}}$ if and only if they hold for $wy \in L_{\mathcal{P}}$, where $y \in (c_2^v)^*$. Briefly, this is a consequence of the fact that $succ(s_0, w) \sim_{P^{\neg c_2}} succ(s_0, y)$ [by Corollary 5.16] and that $h_{c_1^v \cup I^v}(w) = h_{c_1^v \cup I^v}(wy)$ and $h_{I^v}(w) = h_{I^v}(wy)$.

We now move to the main proof. Proof is by induction on $m = |w|$.

Proof for $m = 0$;

We must have $w = \varepsilon$ and so $h_{c_1^v \cup I^v}(w) = h_{I^v}(w) = \varepsilon$. The result is immediate.

Assume true for all words of length $m - 1$. If $w \in (c_2^v)^*$, then result is immediate from $m = 0$ case and the sub-lemma.

If $w = a_1a_2...a_m$, with $a_m \in c_2^v$, then again result holds by induction and sub-lemma.

Otherwise, $a_m \in c_2^v \cup I^v$.

Let $s$ be the state reached by following $a_1a_2...a_{m-1}$, $(s', \overline{s}')$ the state in $\mathcal{A}^{\overline{I^v c_1}}$ reached by following $h_{c_1^v \cup I^v}(a_1a_2...a_{m-1})$. Then $s' \sim_{P^{\neg c_2}} s$ and $succ(\overline{s}_0, h_{I^v}(a_1a_2...a_{m-1})) = \overline{s}'$ by induction hypothesis.

If $a_m \in c_1^v$, then $a_m$ is enabled at $(s', \overline{s}')$ by Lemma 5.15, the fact that $s' \sim_{P^{\neg c_2}} s$, and the rules of construction of $\mathcal{A}^{\overline{I^v c_1}}$, so i) holds [since $h_{c_1^v \cup I^v}(a_1a_2...a_{m-1})a_m = h_{c_1^v \cup I^v}(w)$]. Let $(s'', \overline{s}') = succ((s', \overline{s}'), a_m)$ in $\mathcal{A}^{\overline{I^v c_1}}$ [second component is still $\overline{s}'$ since $a_m \notin I^v$], $s''' = succ(s, a_m)$ in $\mathcal{A}_{\mathcal{P}}$. Then by Lemma 5.12 and rules of construction of $\mathcal{A}^{\overline{I^v c_1}}$, $s'' \sim_{P^{\neg c_2}} s'''$, so ii) holds. Since $h_{I^v}(w) = h_{I^v}(a_1a_2...a_{m-1})$, iii) also holds.

Finally, we must consider the case where $a_m \in I^v$. As $s' \sim_{P^{\neg c_2}} s$ and $a_m$ is enabled at $s$, we see that condition C1) is fulfilled. Now $h_{I^v}(w) = h_{I^v}(a_1a_2...a_{m-1})h_{I^v}(a_m) = h_{I^v}(a_1a_2...a_{m-1})a_m$; i.e. $h_{I^v}(a_1a_2...a_{m-1})a_m \in L_{I^v}$. Thus $a_m$ is enabled at $\overline{s}$ in $\mathcal{A}_{I^v}$, so C2) is fulfilled. Therefore, i) holds. Let $(s'', \overline{s}'')$ be the state of $\mathcal{A}^{\overline{I^v c_1}}$ reached by firing $a_m$; from rules of construction of $\mathcal{A}^{\overline{I^v c_1}}$, $\overline{s}''$ is indeed the state of $\mathcal{A}_{I^v}$ reached by following $h_{I^v}(w)$, so iii) holds. Also, since $s' \sim_{P^{\neg c_2}} s$, we see from Lemma 5.12 that $s'' \sim_{P^{\neg c_2}} s'''$, where $s''' = succ(s, a_m)$ in $\mathcal{A}_{\mathcal{P}}$, so ii) holds.

Hence result. □

**Corollary 5.24.**

$$L_{c_1^v \cup I_v} \subseteq L^{\overline{I^v c_1}}$$

*Proof.* Lemma 5.23 tells us that for all $w \in L_{\mathcal{P}}$, $h_{c_1^v \cup I^v}(w) \in L^{\overline{I^v c_1}}$. Hence result. □

We now come to the most important part of the proof. Recall that during the process of "severing" $C_2$ from $I$ and $C_1$ several base-actions of $I$ have had some of their input/output places removed; in the Petri net framework, this equates to *removing* restrictions on the firing conditions of these actions. How, then, can we be sure that these actions do not fire at an inappropriate time? That is, how do we know that an action in $I^v$ can fire at a certain point in our construction only if it may fire at a corresponding point in the full behaviour (remember, at the corresponding point in the full behaviour, the action may be disabled by some of its input/output places that are not present in the reduced Petri net we are using)? To put it in yet another way, how do we make sure that the construction does not give rise to false states or computations? This is a seemingly serious problem, for we are entirely ignoring the whole component of $C_2$ in our construction, and this component plays a major part in determining whether actions in $I^v$ may fire.

We show now that, perhaps surprisingly, we are fully justified in stating that these actions never fire at "inappropriate" times during our construction.

**Lemma 5.25** (the construction is no more expressive than the full behaviour; that is, "faithfulness" w.r.t $C_1$ holds in the other direction.). *If* $w \in L^{\overline{I^v c_1}}$, *then* $w \in L_{c_1^v \cup I^v}$.

*Proof.* If $w \in L^{\overline{I^v c_1}}$, then $w \in (c_1^v \cup I^v)^*$. The word $w$ may be written as

$$w = x_1 I_1 x_2 I_2 ... x_n I_n x_{n+1}$$

and $h_{I^v}(w)$ may be written as $I_1 I_2 ... I_n$, where in both cases,$n \geq 0$, $I_i \in I^v$, $x_i \in (c_i^v)^*$. By construction of $\mathcal{A}^{\overline{I^v c_1}}$,

$$h_{I^v}(w) \in h_{I^v}(L_{\mathcal{P}}) \tag{5.9}$$

Therefore, $\exists w' \in L^{\overline{I^v c_1}}$ such that $h_{I^v}(w') = I_1 I_2 ... I_n$. We may write $w'$ in the form

$$w' = w_1 I_1 w_2 I_2 .... w_n I_n w_{n+1}$$

where $w_i \in (c_1^v \cup c_2^v)^*$. By an obvious extension of Lemma 5.22, there is a word

$$w'' = x_1'' y_1 I_1 x_2'' y_2 I_2 ... x_n'' y_n I_n x_{n+1}'' y_{n+1} \in L_{\mathcal{P}}$$

where $I_i \in I^v$, $x_i \in (c_i^v)^*$, $y_i \in (c_2^v)^*$.

We claim now that in fact the word

$\overline{w} = x_1 y_1 I_1 x_2 y_2 I_2 .... x_n y_n I_n x_{n+1}$ is in $L_{\mathcal{P}}$. Proof proceeds as follows.

Let $m \geq 0$. Then we claim that:-

i) $x_1 y_1 I_1 x_1 y_2 I_2 ... x_m y_m I_m \in L_{\mathcal{P}}$.

Let $s'$ be the state in $\mathcal{A}_{\mathcal{P}}$ reached by following

$$x_1'' y_1 I_1 x_2'' y_2 I_2 ... x_m'' y_m I_m$$

Let $s$ be the state in $\mathcal{A}_{\mathcal{P}}$ reached by following

$$x_1y_1I_1x_1y_2I_2...x_my_mI_m$$

Let $(s_1, \overline{s})$ be the state in $\mathcal{A}^{\overline{I^vc_1}}$ reached by following

$$x_1I_1x_2I_2...x_mI_m$$

Then

ii) $s \sim_{P^{\neg c_2}} s_1$; and

iii) $s \sim_{P-P^{\neg c_2}} s'$.

Proof is by induction on $m$. The $m = 0$ case corresponds to the words above all being equal to $\varepsilon$, and the result is immediate. So assume true for $m$, and try to prove for $m+1$.

Note that since $x_{m+1}$ is enabled at $(s_1, \overline{s})$ [since $x_1I_1x_2I_2...x_mI_mx_{m+1} \in L^{\overline{I^vc_1}}$], and $s_1 \sim_{P^{\neg c_2}} s$ by hypothesis, then $x_{m+1}$ is enabled at $s$ in $\mathcal{A}_{\mathcal{P}}$, by Corollary 5.18i). Let$(s_2, \overline{s}) = succ((s_1, \overline{s}), x_{m+1})$. Then

$$succ(s, x_{m+1}) \sim_{P^{\neg c_2}} s_2 \tag{5.10}$$

by Corollary 5.18ii) and rules of construction of $\mathcal{A}^{\overline{I^vc_1}}$.

Also, $s'' = succ(s, x_{m+1}) \sim_{P-P^{\neg c_2}} s$ by Corollary 5.14, and $s \sim_{P-P^{\neg c_2}} s'$ by hypothesis and $s' \sim_{P-P^{\neg c_2}} succ(s', x''_{m+1})$ by Corollary 5.14 again, so that

$$succ(s, x_{m+1}) \sim_{P-P^{\neg c_2}} succ(s', x''_{m+1}) \tag{5.11}$$

Therefore, since $y_{m+1}$ is followable from $succ(s', x''_{m+1})$, and using Corollary 5.20ii), we see that $y_{m+1}$ is followable from $succ(s, x_{m+1})$; i.e., so far we know that $x_{m+1}y_{m+1}$ is followable from $s$.

Now

$$succ(s, x_{m+1}y_{m+1}) \sim_{P^{\neg c_2}} succ(s, x_{m+1}) \tag{5.12}$$

by Corollary 5.16. From (5.11), and using Corollary 5.20ii),

$$succ(s, x_{m+1}y_{m+1}) \sim_{P-P^{\neg c_2}} succ(s', x''_{m+1}y_{m+1}) \tag{5.13}$$

Also, using (5.12) and (5.10),

$$succ(s, x_{m+1}y_{m+1}) \sim_{P^{\neg c_2}} succ(s, x_{m+1}) \sim_{P^{\neg c_2}} s_2 \tag{5.14}$$

We must now show that $I_{m+1}$ is enabled at $succ(s, x_{m+1}y_{m+1})$.

Firstly, note that $I_{m+1}$ is enabled at $(s_2, \overline{s})$ in $\mathcal{A}^{\overline{I^v c_1}}$ by the fact that $(s_2, \overline{s})$ is reached by following $x_1 I_1 ... x_m I_m x_{m+1}$ in $\mathcal{A}^{\overline{I^v c_1}}$, and $x_1 I_1 ... x_m I_m x_{m+1} I_{m+1}$ may be followed in $\mathcal{A}^{\overline{I^v c_1}}$ (it is a prefix of $w$). By the rules of construction of $\mathcal{A}^{\overline{I^v c_1}}$, we must have that

$$guard_{I_m,p}\,(s_2(p)) = true \quad \forall p \in (Out_{I_m} \cup In_{I_m}) \cap P^{\neg c_2} \tag{5.15}$$

From (5.14), we deduce that

$$guard_{I_m,p}\,(succ(s, x_{m+1} y_{m+1})(p)) = true \quad \forall p \in (Out_{I_m} \cup In_{I_m}) \cap P^{\neg c_2} \tag{5.16}$$

Since $I_m$ is enabled at $succ(s', x''_{m+1} y_{m+1})$, we must have that

$$guard_{I_m,p}\left(succ(s', x''_{m+1} y_{m+1})(p)\right) = true \quad \forall p \in (Out_{I_m} \cup In_{I_m})$$

In particular,

$$guard_{I_m,p}\left(succ(s', x''_{m+1} y_{m+1})(p)\right) = true \quad \forall p \in (Out_{I_m} \cup In_{I_m}) \cap (P - P^{\neg c_2}) \tag{5.17}$$

In light of (5.13), (5.16) becomes

$$guard_{I_m,p}\,(succ(s, x_{m+1} y_{m+1})(p)) = true \quad \forall p \in (Out_{I_m} \cup In_{I_m}) \cap (P - P^{\neg c_2}) \tag{5.18}$$

Combining (5.15) and (5.17) to get

$$guard_{I_m,p}\,(succ(s, x_{m+1} y_{m+1})(p)) = true \quad \forall p \in (Out_{I_m} \cup In_{I_m})$$

we see that $I_{m+1}$ is indeed enabled at $succ(s, x_{m+1} y_{m+1})$. Thus part i) of the induction hypothesis holds. That $succ(s, x_{m+1} y_{m+1} I_{m+1}) \sim_{P - P^{\neg c_2}} succ(s', x''_{m+1} y_{m+1} I_{m+1})$, [part ii) of induction hypothesis] and $succ(s, x_{m+1} y_{m+1} I_{m+1}) \sim_{P^{\neg c_2}} s_3$ where $(s_3, \overline{s}') = succ((s_2, \overline{s}), I_{m+1})$ [part iii) of induction hypothesis] hold is a consequence of (5.13) and (5.14) respectively, coupled with Lemma 5.12.

So by induction, the word $x_1 y_1 I_1 x_2 y_2 I_2 .... x_n y_n I_n$ is in $L_{\mathcal{P}}$. It is now a simple matter to show that $x_1 y_1 I_1 x_2 y_2 I_2 .... x_n y_n I_n x_{n+1} = \overline{w}$ is in $L_{\mathcal{P}}$. Thus $h(\overline{w})_{c_1^v \cup I^v} \in L_{c_1^v \cup I^v}$. But $h(\overline{w})_{c_1^v \cup I^v} = w$; Lemma proved. □

**Theorem 5.26** (Main result for this section).

$$L^{\overline{I^v c_1}} = L_{c_1^v \cup I^v}$$

*Proof.* Immediate from Corollary 5.16 and Lemma 5.21. □

It is now a simple matter to "abstract away" the unwanted actions in $I^v$, leaving just the required behaviour $L_{c_1^v}$.

### 5.2.2 Improving the Result

In the introduction to this result, we implied that we needed to construct the entire interface-level (or "core"-) behaviour of the system in order to construct the behaviour of $C_1$ & $C_2$, and that we were currently restricted to having just two components beside the interface-level one. A quick glance at the mechanics of the proof, however, shows that the only restriction on the set of actions $c_2$ is that they are independent of all actions in $c_1$. Therefore, we see that in order to construct the behaviour of $C_1$ we may in fact set $I$ to be just the set of actions not in $c_1$ but dependent on actions in $c_1$, and $c_2$ to be every other action, even if the system has the form of more than two components accessing the central, interface-level "hub"; the diagram in Figure 5.4 may help to clarify this.

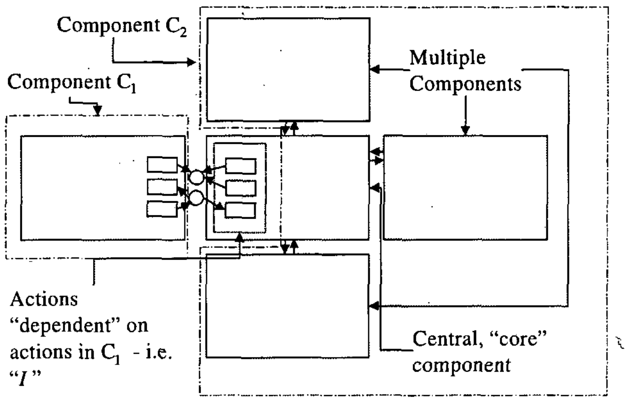

Component C₂
Component C₁
Multiple Components
Actions "dependent" on actions in C₁ - i.e. "I"
Central, "core" component

FIGURE 5.4: The result works for more complex systems than originally implied

Thus, the set $I$ of non-hidden actions is much smaller than we implied (so the partial-order reduction should be correspondingly more effective) and we are not in fact restricted to a 3-component system after all. Note that we may construct the behaviours of several components using just one application of the partial-order reduction by setting $I$ to be the set of actions dependent on actions in any of the required components by symmetry. We may then use this compositional verification result with the resulting automaton to construct the behaviours of the desired components in turn.

We now deal with one obvious objection to the usefulness of the result. Recall that we are constructing the full behaviour of the component $C_1$, $L_{c_1} = h_{c_1}(\mathcal{P})$ by first constructing an $h_I$-compatible partial-order reduced version of $L_\mathcal{P}$ and then using this to construct an automaton $\mathcal{A}^{\overline{I \cup c_1}}$ from which $L_{c_1}$ can be extracted. This raises the question: would it not be easier to simply construct $h_{c_1}$ by means of an $h_{c_1}$-compatible partial-order reduced version of $L_\mathcal{P}$? This is a valid objection *if* the construction of an $h_{c_1}$-compatible partial-order reduced version of $L_\mathcal{P}$ was more efficient (i.e. contained less states) than a $h_I$ -compatible partial-order reduced version. However, the assumptions we have made for this scenario (that $I$ was assumed to be a small interface between the larger components $C_1$ and $C_2$, coupled with the improvements to the results presented in the last section that means we can get away with a smaller $I$ than we had thought) actually make it much more likely that the $h_I$ -compatible partial-order reduced version will be smaller than the $h_{C_1}$-compatible partial-order reduced version of $L_{\mathcal{P}}$, intuitively because there will be a much smaller set of non-hidden actions in the former case.

Recall that an $h$-compatible partial-order reduction is a standard partial-order reduction with some additional constraints on when two actions $a, b \in \Sigma$ are independent - we must have that $h(ab) = h(ba)$, which implies that one or both of $a$ and $b$ are hidden by $h$, or they are both mapped to the same non-hidden image. With a mapping of the form in Definition 5.5 which hides all actions not in $X$ but which is the identity on all actions in $X$, two distinct actions $a, b \in X$ can never be $h_X$-compatibly independent. Since partial-order reduction gives better results the fuller the independence relation, we would want, for an $h_X$-compatible partial-order reduction of a language, the set $X$ to be as small as possible. As per our assumption, the set $X = I$ will be smaller than either of the sets $X = C_1$ and $X = C_2$.

Another reason why one would use this result instead of directly using an $h_{C_1}$-compatible partial-order reduction is that we may want to later construct the behaviour $h_{C_2}$. With our result, this can be easily and efficiently created from the automaton $h_{I^v}$ we used to construct $h_{C_1}$ using a symmetry argument. Without the result presented in this Chapter, we would have to do yet another expensive partial-order reduction of $L_{\mathcal{P}}$, this time with the encumberance of $h_{C_2}$-compatibility.

## 5.3 Summary

In this chapter, we have presented a specific problem: given a Petri-net specified system that can be decomposed into three components labelled $C_1$, $C_2$ and $I$, where $C_1$ and $C_2$ are assumed not to affect each other directly (and where it is further assumed, for reasons that will be re-iterated shortly, that $|I| < |C_1|$ and $|I| < |C_2|$), efficiently construct the full behaviour and local states of $C_1$. A "component" is defined as a subset of base-actions of the Petri net, and from the statement of the problem, $I$ and $C_2$ can be deduced automatically when we have made our choice of $C_1$. One approach is to try to isolate $C_1$ from $C_2$ and attempt to explore its local state-space by paying as little heed to $I$ and $C_2$ as possible, with a view to reducing the total number of global states explored by reducing the number of interleaved local states of $C_1$ and $C_2$ as much as possible. In attempting this, however, we run into the *environment problem*, where restrictions on which actions of $C_1$ can fire at a given point (and hence, which local states the isolated $C_1$ component may enter) are removed by the isolation process, leading to local states and behaviours of the isolated $C_1$ component that have no counterpart in the full system.

By taking a compositional minimisation approach to tackling the enviroment problem, we remedy this by constructing an automaton which acts as a representative of $I + C_2$, behaving exactly like it from the point of view of $C_1$. This representative is created by using an $h_I$-compatible partial-order reduction of the full system, using either the algorithm of Chapter 3 or possibly, if it proves more efficient in general, the algorithm of Appendix B. To simplify the definitions, algorithms and proofs, as well as to ensure that the result is not tied to any particular dialect of Petri nets, *abstract Petri nets* were introduced. These merely remove some dialect-specific features of Petri nets and extract some common features.

Having presented the algorithm for constructing the full behaviour and local state-space of $C_1$, we then showed that the result not only enables us to construct the full behaviour of $C_1$, but also that of $C_2$ by symmetry, both based on the same $h_I$-compatible partial-order reduction of the full system. Further, the definition of "$C_1$" was revealed to be somewhat arbitrary: "$C_1$" may encompass several components of the system, as long as we choose $I$ and $C_2$ so that $C_2$ does not affect any of the components in $C_1$. Thus, the result allows us to reconstruct the full behaviours and state-spaces of multiple components, all from the same $h_I$-compatible partial-order reduction.

There appears to be a much simpler method of achieving our original goal of constructing the full behaviour of $C_1$ than this rather circuitous approach, however: simply computing the $h_{C_1}$-compatible partial order reduction of the full system would suffice. If this simple approach is just as effective as the one we have developed in this chapter, it could undermine the usefulness of this result. We argue, however, that the assumption that $|I| < |C_1|$ would imply that the $h_I$-compatible partial order reduction would be much more efficient to construct than the $h_{C_1}$-compatible one. Recall that the effectiveness of partial-order reduction is dependent on the number of pairs of actions that are mutually independent: generally, having a large proportion of pairs of actions being independent will yield large reductions in the size of the state-space, whereas having a small proportion will give lesser reductions. Since an $h_X$-compatible partial order reduction requires all actions with base-action $X$ to be mutually dependent, it follows that we would wish $X$ to be smaller. The additional benefit of being able to construct $C_2$ from the same partial-order reduction is another advantage of the result presented in this section over this alternate approach.

Chapter 6

# A Practical Example - The Sliding Windows Protocol

## 6.1 Overview

In this chapter,we present a brief "proof of concept" for the result of Theorem 3.6 to show that a useful level of state-space reduction can be achieved with partial-order reduction, even with the restriction of $h$-compatibility imposed. We would like to stress that the prototype tool we will use is still very preliminary, and several of the required algorithms are not yet fully implemented, in particular, the generalised check for weak continuation-closure (see Appendix A), minimisation of automata, and the check for satisfaction within fairness (see Appendix D). Also, the implementation of the stubborn set construction algorithm is extremely primitive.

The system we will be verifying is a simple variant of the *Sliding Windows Protocol* [Holzmann (1991)] (in particular, it is a modification of the variant known as the *Go-Back-n Protocol*, expressed in the Petri Net formalism presented in Chapter 4). Although the stubborn set implementation is not at all advanced, it does include the queue-based optimisations presented in the same Chapter.

We will present, for a number of configurations of our Sliding Windows Protocol implementation, the number of states in the state-space of the full system, the number of states resulting from an ordinary partial-order reduction of the system and then, to see how much difference $h$-compatibility makes to the reduction obtained from partial-order reduction, the number of states resulting from a partial-order reduction of the system with three different approaches to incorporating $h$-compatibility, including that suggested by the result of Theorem 3.6.

## 6.2 The Sliding Windows Protocol

The Sliding Windows Protocol allows for sequences of messages to be sent and subsequently received, in the correct order, over a lossy channel that may lose, duplicate or re-order messages sent over it. It affords greater efficiency in terms of utilisation of available bandwidth than the less sophisticated method of waiting for an acknowledgement of the previous sent message before sending the next (the *Stop-and-Wait Protcol*) by allowing more messages to be in transit between acknowledgement receipts. The basic method is as follows; we will describe only the *Go-Back-n* variant.

The sender maintains a "window" consisting of a consecutive sequence of *frames*, numbered between *frameBegin* and *frameEnd*, say. The size of the window (the difference between *frameBegin* and *frameEnd*) is initially zero. Whenever the current size of the window is less than the maximum size, *WindowSize*, the sender may send the next message in his queue. Whenever a message is sent, it is tagged with the current value of *frameEnd*, and the value of *frameEnd* is incremented, incidentally increasing the size of the current window. The following mechanisms described are designed to ensure that the following conditions hold: that every message sent by the receiver is tagged with a frame number; that no two messages share a frame number; and that at any time, all messages in the current window (that is, all messages tagged with a frame between *frameBegin* and *frameEnd* $- 1$, inclusive) are those messages that have been sent but which have not yet been acknowledged as having been received by the receiver in the correct order. Any message and frame pair in the current window is eligible to be re-sent at the sender's discretion; typically, a message will be re-sent after a certain period has elapsed since it was last sent, and no acknowledgement has been received.

The receiver maintains a value called *ExpectedTag*, representing the frame number tagged onto the next message it expects to receive. This number is initially chosen to correspond to the frame tag given to the first message the sender will send, and is agreed upon by sender and receiver upon initiation of the connection. At any time, any message retrieved by the receiver from the communication channel which is *not* tagged with the expected frame number is discarded; this is to ensure that the receiver does not accept a message that was dispatched *after* the required message, which would be to tantamount to receiving the messages sent by the sender in the wrong order.

When a message tagged with the expected frame is received, an acknowledgement is sent back by the receiver to the sender, consisting of just the frame number. The expected frame number is then incremented. As with sent messages, the acknowledgements are not safe from being lost, duplicated or re-ordered while in transit.

When the sender receives an acknowledgement, it is checked to see whether it is equal to the current value of *frameBegin* (and thus corresponds to an acknowledgement of receipt of the earliest sent message that has not yet been properly acknowledged), and if so, the current value of *frameBegin* is incremented (decreasing the size of the current window and allowing more messages to be sent, and removing the acknowledged messages from the list of tagged messages that might need to be resent); otherwise, the acknowledgement is simply discarded, as we cannot know for sure that the receiver has received this message, and received it in its correct place in the sequence of messages sent by the sender.

The process continues, with the window gradually "sliding" as the values of *frameBegin* and *frameEnd* steadily increase (roughly in step with that of *ExpectedTag*) and the messages in the current window being resent until they are acknowledged as having been received in the desired order, until the whole sequence of message has been correctly sent and received (at which point the window will have zero size, and there will be no more messages to add to it).

## 6.3 Petri Net Specification

Our current Stubborn Set implementation (see Chapter 4) relies on the fact that the domains of each place (that is, the set of tokens that the place is allowed to contain) are finite and preferably small, which would not be the case when we are sending large messages with the protocol just described: with long messages, *frameBegin*, *frameEnd* et al would necessarily assume large values, and so the places containing these values would need correspondingly large domains. Instead, we have opted for a slightly unusual variant in which the *frameBegin*, *frameEnd* etc values are bounded, and the specification is designed such that should any of these variables exceed their bounds, they "wrap-around" back to the value 1, creating what might be called a "cyclical-window" protocol[1]. Now, one of the conditions that the Sliding Windows specification takes care to ensure is that no two messages are ever labelled with the same frame number, which is clearly not the case here, and it so happens that this condition is essential in order to ensure that the messages are received in the order they are sent, even when the channel may duplicate and re-order messages sent through it. Thus, this new protocol will *not* have the desired properties when sent over a channel that may re-order and duplicate messages, although, as we will show in this Chapter, it will work if the channel is allowed only to lose messages.

The Petri Net specification of this protocol is as shown in Figure 6.1. The preamble defines three constants: *NumMessageObjects*, *WindowSize* and *MaxInQueue*, which allows us to parameterise the protocol. The specification models a sender that sends sequences of messages drawn from a pool of *NumMessageObjects* possible messages (each coded for convenience as a number in the range $[1, \mathit{NumMessageObjects}]$) in arbitrary order

[1]Actually, this restriction is not all that unrealistic as TCP (for example) uses the same approach, albeit with a much higher upper-bound before "wrapping" occurs.

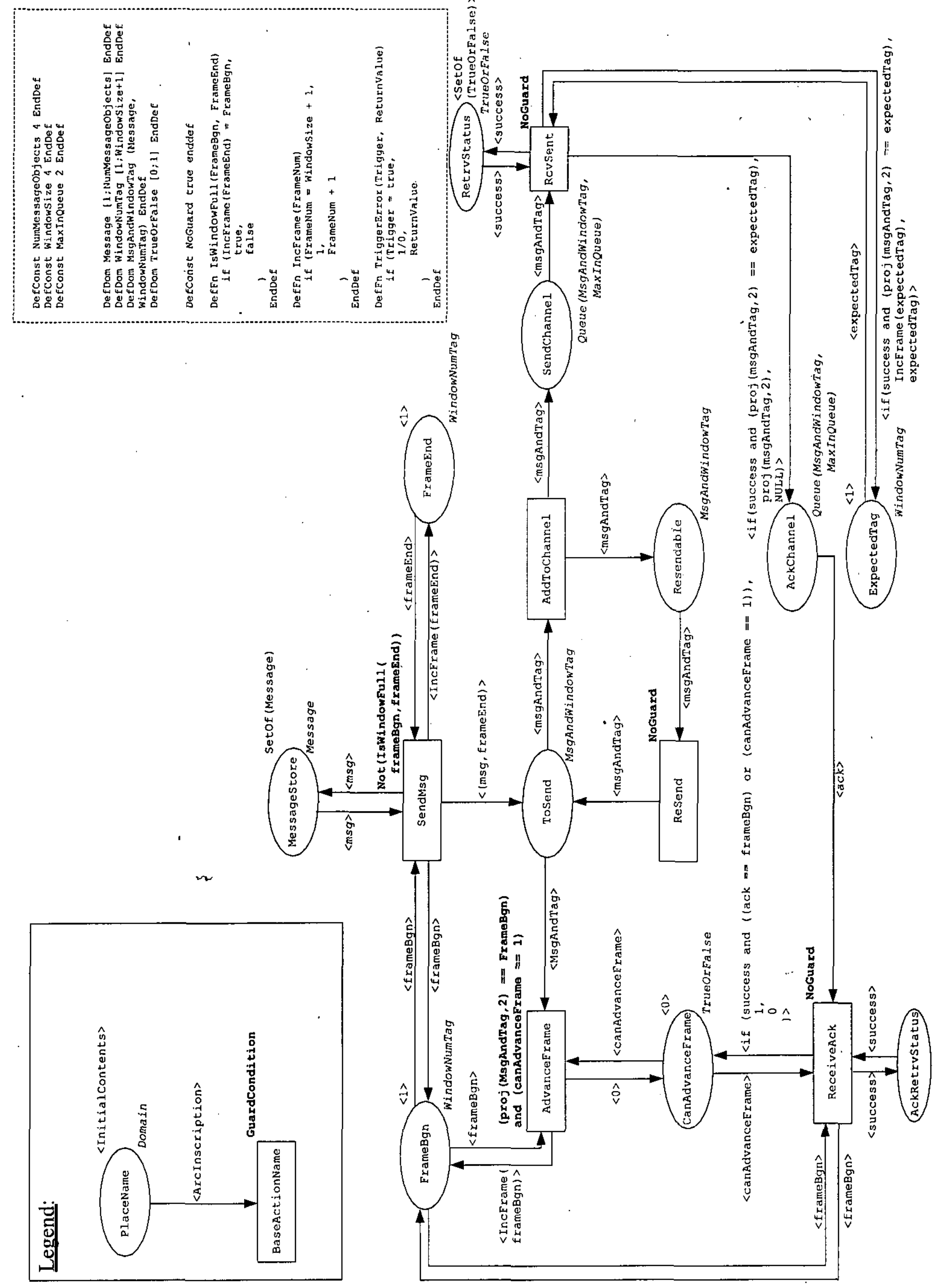
and with arbitrary sequence length (in fact, the sender modelled here simply never terminates). The protocol specification has been designed with simplicity in mind and as such contains shortcuts and omissions: for example, the protocol does not wait until a certain length of time has passed before a message is resent, as would a "real-life" specification - a message is instead eligible for re-sending as soon as it is sent i.e., as is usual in protocol modelling, non-determinism is used to model time-outs. A more detailed description of the specification follows.

The possible values for a frame number (e.g. the variables *frameBegin*, *frameEnd* and *expectedTag*) range from 1 to *WindowSize* + 1, inclusive. Initially, both *frameBegin* and *frameEnd* are set to the same value (signifying an empty window), which we have chosen as 1. The preamble defines a pair of "convenience" functions: *IncFrame*, which returns the value that the provided frame number would attain if it were incremented (including "wrapping-around", if necessary), and *IsWindowFull*, which tests whether or not the current window (as defined by the values of *frameBegin* and *frameEnd*) is full, and correspondingly whether any new messages may be sent.

The base-action (see Definition 4.6 and commentary) *SendMsg* has the following variables: *msg*, *frameBegin*, and *frameEnd*. If the guard condition (that the current window is not full) is fulfilled, the tagged message (consisting of the pair $(msg, frameEnd)$) is added to the set of tagged messages to be sent over the channel, and the current value of *frameEnd* is incremented. The *ToSend* device is used to simplify the addition of a "re-sending" mechanism to the protocol, and is purely an implementation detail. The base-action *AddToChannel* takes, when the communication channel *SendChannel* is not full, a tagged message from the current list of tagged messages to be (re-)sent and pushes it into the channel, which is modelled by a bounded queue. It also adds the tagged message to the "re-sendable" set of tagged messages.

*RcvSent* simulates the retrieval of a tagged message *msgAndTag* by the receiver from the *SendChannel*, or possibly the loss of this tagged message by the channel during transference, depending on the value of success (true or false; both are possibilities whenever a message is received). If, and only if, the frame number tagged onto the message (since tagged messages are of the form $(msg, frameNum)$, this value is equal to $proj\ (msgAndTag, 2)$) is equal to *expectedTag and* the message was received successfully (i.e. was not lost in the channel, corresponding to the condition *success* equals 1), then an acknowledgement consisting of the frame number tagged onto the received message is sent, and the value of *expectedTag* is incremented. Under any other cirumstances, the retrieved tagged message is discarded.

*RcvAck* simulates the receipt of an acknowledgement from the receiver (in the form of the frame number tagged on to the message received). As with *RcvSent*, there is a *success* variable simulating the lossiness of the communication channel. If, and only if, the retrieval of the acknowledgement is successful and the acknowledged frame matches


```
DefConst NumMessageObjects 4 EndDef
DefConst WindowSize 4 EndDef
DefConst MaxInQueue 2 EndDef


DefDom Message [1;NumMessageObjects] EndDef
DefDom WindowNumTag [1;WindowSize+1] EndDef
DefDom MsgAndWindowTag (Message,
WindowNumTag) EndDef
DefDom TrueOrFalse [0;1] EndDef


DefConst NoGuard true enddef

DefFn IsWindowFull(FrameBgn, FrameEnd)
    if (IncFrame(FrameEnd) = FrameBgn,
        true,
        false
    )
EndDef

DefFn IncFrame(FrameNum)
    if (FrameNum = WindowSize + 1,
        1,
        FrameNum + 1
    )
EndDef

DefFn TriggerError(Trigger, ReturnValue)
    if (Trigger = true,
        1/0,
        ReturnValue
    )
EndDef
```

FIGURE 6.1: The Sliding Windows Protocol

the value of *frameBegin*, then the *CanAdvanceFrame* flag is set to true.

*AdvanceFrame* handles the advancement of the start of the window, *frameBegin*, when an acknowledgement of receipt of the first message in the window is received. It can fire if and only if *CanAdvanceFrame* is true, and if the (unique) message in the current window tagged with the current value of *frameBegin* is in the "to (re-)send" list of tagged messages; the latter condition is an (admittedly clumsy) means of ensuring that this successfully received message is not re-sent[2].

This describes the basic workings of the protocol; we now describe the abstracting homomorphism that we will use, and which properties it will allow us to decide are satisfied within fairness by the full behaviour, in conjuction with Theorem 3.6.

The abstraction $h : \Sigma \rightarrow \Sigma' \cup \{\varepsilon\}$ defined one the set $\Sigma$ of all possible actions hides all actions, unless the action $a$ fulfils one of the following conditions:

- the action $a$ has *SendMsg* as its base-action. In this case, the action is mapped to "Send*msg*" i.e. the value of the variable *msg* in the binding of the variables associated with *SendMsg* comprising the action $a$, prefixed by the word "Send"; or
- the action $a$ has *RcvSent* as its base-action, the value of *success* in the binding to the variables associated with *RcvSent* is true, and the value of the tag is equal to the value of *ExpectedTag*; in other words, the conditions for a message to be successfully received (and subsequently acknowledged). In this case, the action is mapped to "Rcv*msg*" where $msg = proj\,(msgAndTag, 1)$ i.e. the message part of the variable *msgAndTag*, prefixed by the word "Rcv".

Thus, the abstract set of actions $\Sigma'$ consists of the set of actions labelled

$$\{\text{Rcv1}, \text{Rcv2}, ..., \text{Rcv}M\}$$

together with the set

$$\{\text{Send1}, \text{Send2}, ..., \text{Send}M\}$$

where $M$ is the number of different messages that can be sent through the system (it is just equal to the parameter *NumMessageObjects*, used instead only because it is shorter).

This abstracting homomorphism is sufficient to decide, using the result of Theorem 3.6, whether the full behaviour encoded by the language $L \subseteq \Sigma^*$ satisfies (within fairness) the property "whenever a message $m$ is sent, it is eventually received" based only on the abstract automaton representing the language $h(L)$, computing from the $h$-compatible trace reduction obtained (assuming, of course, that $h$ is WCC on $L$); it is true if and only if

$$h(L) \models_{WF} \mathcal{P}$$

where $\mathcal{P}$ is the property expressed in PLTL as

[2]In the original specification, this was an optional optimisation to avoid having to resend redundant messages; in this "cyclical" variant, however, it is actually necessary for the protocol to function correctly.

$$G(\text{Send1} \implies F(\text{Rcv1})) \wedge G(\text{Send2} \implies F(\text{Rcv2})) \wedge ... \wedge G(\text{Send}M \implies F(\text{Rcv}M))$$

Perhaps more interestingly, we may also verify the property that all messages are received in the correct order by taking as our property $\mathcal{P}$ all sequences $w$ of actions in $\Sigma'^*$ for which, if we ignore the Rcv actions to get the subsequence

$$\text{Send}a_1\text{Send}a_2\text{Send}a_3...$$

consisting of all Send$m$ actions of $w$, in order of occurrence in $w$, and then ignore the Send actions to get the subsequence

$$\text{Rcv}b_1\text{Rcv}b_2\text{Rcv}b_3...$$

of Rcv actions in order of occurrence in $w$, satisfies $a_i = b_i$ for all $i$. Although this property probably cannot be expressed as a PLTL formula, it is not probably too difficult to construct an algorithm for constructing an automaton representing this property i.e. an automaton that encodes the language $pre(\mathcal{P}) \subseteq \Sigma^*$. The full behaviour $L$ then satisfies *linearly* [Alpern and Schneider (1985)] the property that all messages sent are received in the correct order if and only if $h(L) \subseteq pre(\mathcal{P})$. Informally, this is because the full behaviour does not satisfy the property if and only if there is a trace $w$ in which the messages are received in the wrong order. Since the abstracting homomorphism $h$ preserves all "Send" and "Receive" events, this is true if and only if, in the abstract trace $h(w)$, the messages are received in the wrong order which, by choice of $\mathcal{P}$, is true if and only if $h(w) \notin \mathcal{P}$.

## 6.4 Experimental Results and Commentary

In this section, we present a short series of practical results, summarised in Table 6.1. The experiment consisted of computing, for a small sample of values for each of *NumMessageObjects*, *WindowSize* and *MaxInQueue*, the following automata from our example Sliding Windows Protocol system:

- *Unreduced*: the full state space of the system;
- *hNONE*: a partial-order reduced version of the system without the encumberance of $h$-compatibility;

| $M$ | $W$ | $Q$ | Unreduced | $h$NONE | $h$SOFSEM99 | $h$VCL00 | $h$PODC01 |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 2 | 88260 | 30188 | 84910 | 44380 | 33627 |
| 3 | 2 | 2 | 12670 | 4122 | 12247 | 6008 | 4664 |
| 3 | 2 | 3 | 38675 | 11212 | 37168 | 19611 | 12128 |
| 3 | 2 | 4 | 112186 | 29765 | 107774 | 55558 | 33075 |
| 3 | 2 | 5 | 312423 | 74369 | 299844 | 116550 | 81504 |
| 3 | 3 | 2 | 342256 | 116119 | 328833 | 157619 | 120925 |
| 3 | 3 | 3 | N/A | 452872 | N/A | 678465 | 471774 |

($W$ = *WindowSize*, $M$ = *NumMessageObjects*, $Q$ = *MaxInQueue*)

TABLE 6.1: Results of the partial-order reduction on the specification of Figure 6.1

- *hSOFSEM99*: an $h$-compatible partial-order reduced version of the system where actions $a$ and $b$ cannot be independent unless $h(a) = h(b)$, as in [Ultes-Nitsche (1999)];

- *hVCL00*: an $h$-compatible partial-order reduced version of the system where actions $a$ and $b$ cannot be independent unless $h(ab) = h(ba)$, and an $h$-compatible persistent set must contain at least one action that is not hidden by $h$, as in [Ultes-Nitsche and St James (2000)]; and

- *hPODC01*: an $h$-compatible partial-order reduced version of the system where actions $a$ and $b$ cannot be independent unless $h(ab) = h(ba)$, and $h$-compatible persistent sets may be completely hidden by $h$, as in [St James and Ultes-Nitsche (2001)] and Theorem 3.6 of this thesis.

Of these, the most important figures for assessing the usefulness of Theorem 3.6 are the full state-space ("Unreduced") size, the ordinary partial-order reduced ("$h$NONE") size, and the result obtained from an application of Theorem 3.6 ("$h$PODC01"), which together show how much the imposition of $h$-compatibility reduces the reduction obtained.

As we can see, the original attempt at incorporating partial-order reduction ($h$SOFSEM99 in the table) gives very little in the way of reduction, and so is of almost no practical use. This particular result came as no surprise to us: checking each state in the example of Figure 3.1 for persistent sets when the independence relation requires actions to have identical abstract images (the simplicity of that particular system makes finding optimal persistent sets for a state a simple matter of trial and error) revealed that *no* non-trivial persistent sets exist, at any of the 28 states of the system. It seems that the requirement of identical abstract images is too restrictive in practice to be workable.

The successor to this result, where independent actions need only be compatible under the abstraction rather than identical but where completely hidden persistent sets were disallowed ($h$VCL00 in the table), fared much better and appears to be of at least some practical use. The successor to this result where completely hidden persistent sets are now permitted ($h$PODC01 in the table; the result of the main theorem of this thesis) fared better still, although we were surprised at the comparatively small gap between it and its predecessor, $h$VCL00; this appears to result from the fact that this particular example system does not have a particularly large scope for completely hidden persistent sets.

The results, considering the lack of optimisations to the stubborn set algorithm and the lack of parallelity in the chosen example specification, are quite reasonable. In particular, the encumberance of $h$-compatibility, which could conceivably have greatly lessened the reduction, is quite insignificant, at least when using the result of the main theorem of this thesis, Theorem 3.6. One slightly disappointing observation stems from the results of increasing the queue size while keeping the other parameters (the number of message objects and the window size) static; we had hoped that the "queue" optimisations of Section 4.3.4 would have greatly limited the increase in the state-space under partial-order reduction (keeping the increase fairly close to zero, in fact) which unfortunately appears not to have been the case.

Another disappointing observation is that the addition of partial-order methods to the original technique (involving straight abstraction of the full state-space) could make the check for weak-continuation closure less efficient. This is because, based on the example tried here, the partial-order reduction cannot be guaranteed to preserve the strongly-connectedness of the resulting automaton; that is, even when the automaton representing $L$ is strongly-connected, it is not necessarily the case that $R_L^{\Delta}$ is. Since strong-connectedness of the automaton representing $L$ is an efficiently checkable (linear in the size of the automaton being checked) condition that is sufficient to prove that $h$ is WCC on $L$, applying partial-order reductions means that we may lose this efficient means of deciding WCC. However, the result of Appendix A will hopefully lessen the impact of this significantly.

## 6.5 Summary

In order to gauge the usefulness of the result of Theorem 3.6, we have specified a variant of the Sliding Windows Protocol for use with our (currently very primitive) Petri Net tool. The current stubborn set implementation relies on the domains of the Petri net being small and finite, so we have chosen to implement a "cyclical windows" protocol. Our choice of $h$ was such that a couple of interesting properties of the full behaviour could be deduced based on just the abstract behaviour, although only one of these actually utilised the full Theorem 3.6. We calculated the full state-space of the system, a partial-order reduced version ignoring $h$-compatibility, and partial-order reductions using three different definitions of an $h$-compatible persistent set drawn from [Ultes-Nitsche (1999)], [Ultes-Nitsche and St James (2000)] and [St James and Ultes-Nitsche (2001)].

The results were not unacceptable, considering the fact that the specification is not particularly amenable to partial-order reduction. For us, by far the most important aspect of the results is whether the loss of reduction due to the encumbrance of $h$-compatibility is significant as compared to the unencumbered $h$NONE: this is important as generally, partial-order reduction has been a successful method of state space reduction, so the usefulness of the result of Theorem 3.6 depends largely on whether the restriction of $h$ -compatibility greatly hampers the reduction as compared to unencumbered partial-order reduction. We see that the original attempt at incorporating $h$-compatibility, $h$SOFSEM99, predictably hampered the reduction quite severely, rendering this approach almost useless, practically. The successor to this attempt, $h$VCL00, was dramatically more successful, giving a reduction significantly closer to that achieved by $h$NONE. The final attempt, utilising the main result of this thesis, did better still, in general leading to state-spaces only a few percent larger than those resulting from $h$NONE for the larger values of *NumMessageObjects*, *WindowSize* and *MaxInQueue*. Based on this one example, it seems that $h$-compatibility need not significantly hamper the reduction as compared to an unencumbered partial-order reduction, which is a very encouraging result.

On the negative side, it was found that partial-order reduction may not preserve the strong-connectedness of the full state-space of a system, and so by applying partial-order reduction we may lose an efficient check for WCC, forcing us to rely on the less efficient and more complex result of Appendix A. Additionally, the hypothesised gains in reduction from the Queue Emptying Strategy of the Chapter 4 appear to be non-existent.

# Chapter 7

# Conclusions

## 7.1 Summary

Chapter 2 introduced satisfaction of a property by a system *within fairness* (informally, a system satisfies a property within fairness if and only if, no matter what has occurred in the system so far, the system can still progress in a way that satisfies the property) and *abstracting homomorphisms*, which are abstractions performed on the *actions* of a concrete system. Previous work has shown that given a system that can be expressed as a *finite* LTS and a property expressible as a set of $\omega$-words over the set of actions in the abstract system (e.g. a property specified using PLTL), such an abstracting homomorphism $h$ can yield smaller, simpler behaviours while preserving sufficient information to allow us to infer properties satisfied within fairness by the original, concrete system, provided that $h$ is *weakly continuation closed* on the concrete system: i.e. a weakly-continuation-closed abstracting homomorphism *preserves* satisfaction within fairness.

However, the need to a) construct the abstract behaviour itself and b) to prove that $h$ is weakly-continuation closed on the concrete behaviour meant that we were still required to construct the whole (prohibitively large) state-space prior to the abstraction step, reducing the usefulness of this approach. The focus of this thesis has been on improving the practicality of this technique by removing this requirement.

*Partial-order methods*, in particular the persistent set selective search techniques of Godefroid and Wolper, appeared to be a promising approach for achieving this aim: they give a means of reducing the state-space of a system by attempting to avoid following as many redundant interleavings of "independent" actions as possible, which can lead to significant reduction of the state-spaces of systems. However, the persistent set selective search needed to be modified to fit with our goals and the modifications had a price: some interleavings of independent actions that would otherwise be ignored by the persistent set selective search turn out to be crucially important with respect to preserving satisfaction within fairness under $h$, and so the definition of "independence" of actions had to be restricted to ensure *abstraction compatibility* with $h$, and the "sleep-set" optimisation for reducing the number of transitions (but not states) can no longer. be used. Since the persistent set selective search achieves its best reduction when there are many pairs of independent actions, this restriction could negatively impact the reduction obtained from the persistent set selective search, especially if the abstracting homomorphism does not hide many actions.

The definitions of abstraction compatibility went through three progressively less restrictive revisions, and Chapter 3 proved the main result of this thesis: that when the independence relation is modified in line with the least restrictive of these, the resulting partial-order reduction of the concrete state-space can be used entirely in place of the concrete state-space that we were previously required to use. So constructing the concrete state-space was now shown to be unnecessary, but whether or not the replacement $h$-compatible partial-order reduction was of more practical use still required justification.

Since the persistent set selective search is already well-established, especially when dealing with systems with a large amount of pairs of independent actions, e.g. those composed of loosely-coupled components, the most important factor to consider when gauging the practicality of this new method is to what degree the added restrictions of abstraction compatibility affect the reduction obtained from the persistent set selective search. A series of experiments were performed to measure this.

Our practical experiments used Petri nets as the means for specifying systems as they work particularly well with persistent sets. Initial results were not promising: even the most relaxed definition of abstraction compatibility led to significantly larger persistent sets compared to those obtained when abstraction compatibility was ignored, leading to correspondingly larger state-spaces. We explored just why the existing persistent set algorithms were performing so poorly, and modified the algorithms used to account for the special complications arising from $h$-compatibility: essentially, the persistent set construction algorithms involve choosing actions to add to a candidate persistent set at each stage, and non-hidden actions must be heavily discriminated against when choosing these actions. Chapter 4 describes the formulation of Petri nets used for our practical example; our analysis of the consequences of $h$-compatibility on persistent sets; and the persistent set construction algorithms we eventually used in our practical experiments, obtained by modifying existing algorithms to take this analysis into account, and also incorporating the addition of the queues required by the sliding windows example to our Petri net formulation.

For the practical experiment we constructed a parameterisable implementation of a variant of the sliding windows protocol using our formulation of Petri nets, and examined the state-space resulting from: a standard state-space exploration; a standard persistent set selective search without the restriction of $h$-compatibility; and persistent set selective searches using the three revisions of $h$-compatibility. The full analysis is found in Chapter 6 but briefly, it was found that the impact of $h$-compatibility was quite small, with the state-spaces being only a few percent larger than those obtained when $h$-compatibility was ignored. So, in this case at least, the full concrete state-space originally required for constructing the abstraction and deciding whether $h$ was weakly-continuation closed can indeed be replaced by a significantly smaller reduced version.

On a less positive note, it was found that one of the most useful efficiently checkable conditions for deciding whether an abstracting homomorphism is weakly continuation-closed (if the LTS representing the system is strongly-connected, then *all* abstracting homomorphisms are weakly continuation-closed on it) might be rendered less useful: the partial-order reduction does not necessarily preserve strong-connectedness. The less efficient (but more efficient than previous means of deciding weak continuation-closure in the general case) result of Appendix A was created as a fallback.

Chapter 5 presented a result that combined abstraction-compatible partial-order reduction and compositional verification, allowing us to construct the full behaviour of a required set of components without constructing the full behaviour of the system by using a suitable abstracting homomorphism to create an *interface automaton* that represents the behaviour of the full system from the point of view of the required components. This automaton can then be used in conjunction with an algorithm to circumvent the environment problem by allowing us to guide the construction of the required components even when isolated from the rest of the system. The result presented is not the most direct way of using abstraction-compatible partial-order reductions to construct the full behaviour of specific components, but the abstracting homomorphism is chosen in such a way that it should give better partial-order reduction that the more direct means. This hypothesis remains to be experimentally justified.

## 7.2 Related Work

The concept of satisfaction within fairness of a property $\mathcal{P}$ by a behaviour $B$ is a rephrasing "$\mathcal{P}$ is a *relative liveness* property of $B$", emphasing that the satisfaction of a property is considered relative to the possible behaviours of the system. Relative liveness properties were developed in the context of real-time systems in [Henzinger (1992)], although this work was more focussed on its counterpart, relative safety. Relative liveness properties have their roots in the concept of *machine-closure*[Abadi and Lamport (1988, 1990); Alur and Henzinger (1995)], to which they are nearly identical. It was shown in Appendix E that satisfaction within fairness is more "lenient" than satisfaction under strong fairness in that if a behaviour satisfies a property under strong fairness, it will satisfy that property within fairness.

Our main result uses *behaviour abstraction* as its abstraction approach. This differs from data abstraction in that it aims to reduce domain of actions (i.e. labels on the LTS) of the system, rather than reducing the domains of variables that comprise a system state. This approach has a few disadvantages compared to data abstraction: as mentioned in Section 2.4.1.1, whereas the explicit mapping between concrete and abstract states granted by data abstraction allows us to perform the state-space exploration using the smaller set of abstract states only, often with the result that an exploration terminates quite rapidly, with behaviour abstraction the set of abstract states cannot be deduced until *after* an exploration of the full state space (or, in the light of the main result of this thesis, an abstraction-compatible partial-order reduction of this) has been completed, so data abstraction can generally be relied upon to finish more quickly. Our behaviour abstraction approach is also unable to deal with infinite state-spaces (see Appendix C) and, crucially, is unable to deal with properties specified in CTL.

In its favour, the main result of this thesis is quite simple and general, and has a broad applicability: for example, systems whose behaviour can be represented by a finite, strongly-connected LTS are quite common, and we can state that for *all* such systems (i.e. we do not care about the specific means of specifying the system used: it could be Petri net-based, PROMELA, etc), for *all* abstracting homomorphisms $h$ (i.e. we do not care about the specific abstraction used), and *any* property that can be expressed as a set of $\omega$-words over the range of $h$, the abstract behaviour satisfies the property within fairness if and only if the concrete behaviour satisfies the concrete version of the property. Data abstraction may need more tailoring to the specific formulation and abstraction used[J. Dingel and T. Filkorn (1995)]. Also, in the field of communications where we may be more interested in the presence/ order of messages sent through the system, behaviour abstraction seems to be a better fit: simply use an abstraction that preserves those messages that we are interested in and hides the rest of the system.

The main result refines the previous attempts to marry partial-order reduction and satisfaction within fairness under abstracting homomorphisms in [Ultes-Nitsche (1999); Ultes-Nitsche and St James (2000)] by reducing the restrictions on what it means for a pair of actions to be $h$-compatibly independent; a comparison of the practical differences arising from relaxing these restrictions is given in Chapter 6.

The result of Chapter 5 attempts to tackle just the *environment problem*[McMillan (2000, 1997)] in the field of compositional verification, and does not attempt to propose decomposition strategies [Nam and Alur (2006); Cobleigh et al. (2003); Cobleigh et al.] nor reasoning about properties of the complete system based on verified properties of the system's components[Misra (2001); Cohen (2002); Ehmety and Paulson (2005)]. There are two main approaches to the environment problem: *assume-guarantee*[Jones (1983); Pnueli (1985)], and *compositional minimisation*[M. Chiodo et al. (1992); de Alfaro and Henzinger (2001); Liu (2000)]. We wanted to create the *full* behaviour of a component $C_1$, and while there are automated techniques for generating assumptions[Cobleigh et al.; de la Riva and Tuya (2006)], they don't necessarily scale well[Shankar (1998); de la Riva and Tuya (2006)], there may be circular reasoning problems when verifying the assumptions themselves[Barringer and Giannakopoulou (2003)], and encoding the full behaviour of the remainder of the system as a set of assumptions may not be feasible. For these reasons, we opted for the compositional minimisation approach, using an abstraction compatible persistent set reduction of the system to create the surrogate component representing the behaviour of the rest of the system from the point of view of $C_1$.

Probably the closest result to that of the one in Chapter 5 (in terms of capability, if not approach; the approach of using an abstraction compatible persistent set reduction of the system as the surrogate component appears to be novel) is that of [Christensen and Petrucci. (2000)]. This aims to construct the full behaviour of a component by exploiting the structure of the system to explore all components separately as far as possible, using a "synchronisation graph" to remove the disadvantages imposed by the environment problem, which corresponds roughly to the $\mathcal{A}_{I^v}$ automaton. This approach makes more effective use of the structure of the system and reduces a lot of the interleaving of local states, but doesn't reduce the amount of interleavings *within* a component as is the case with our persistent set approach. The paper does not target high-level Petri nets, but notes that generalising to these should be easy - in addition, there seems to be nothing prevent it working with the abstract Petri nets that our result works with. Attempting to combine the two approaches might be an interesting topic of future research.

## 7.3 Future Work

Much of the future work involves experimental justification of some of the results presented in this thesis. Testing the results of Chapter 5 is also a priority, as both results may well be less efficient than other techniques. Attempting to combine the result with that of [Christensen and Petrucci. (2000)] may also be fruitful. A comparison of Algorithm 3.1 and Algorithm B.1 to see which performs better when we wish to compute the abstraction of a behaviour would also be interesting. It would also be interesting to see if the main result of this Thesis could be adapted to work with *ample sets*[Peled (1994)] in addition to persistent sets, especially in the light of efficient ample-set construction techniques such as those in [van der Schoot and Ural (1996)].

# Appendix A

# An Improved Decidability Result for WCC

## A.1 Overview

In this Appendix, we present an algorithm for deciding whether a homomorphism $h$ is weakly continuation-closed on a language $L$. The starting result is a theoretical result by Ochsenschläger [Ochsenschläger (1992)] which proved decidability of WCC by showing that the problem reduced to deciding a simpler condition on pairs of states (no explicit algorithm was provided) from $\mathcal{A}_L$ and $\mathcal{A}_{h(L)}$. We refine this result by drastically reducing the number of pairs of states that must be checked before going on to provide an algorithm for deciding the aforementioned simple condition. It should be noted that the decision algorithm is still not 'efficient' since we must determinise sub-automata of the automaton representing the concrete behaviour, and we would conjecture that there does not exist such an efficient decision algorithm.

The reasons why this result is hidden away in an Appendix rather than in the main body of the thesis are twofold; firstly, there exist efficiently checkable sufficient conditions for WCC that are often satisfied by most 'realistic' systems; and secondly, if a system does not satisfy these conditions it is very unlikely that a given homomorphism will be WCC on it.

## A.2 Preliminaries

In this section, we will re-cap some old definitions, present some new ones, and prove some preliminary Lemmas and Theorems, before moving onto the new algorithm.

Let $L$ be a prefix-closed regular language represented by the (minimal) LTS $\mathcal{A}_L$, which contains no deadlocks; if a deadlock state $s$ exists in $\mathcal{A}_L$, we add a transition from $s$ to itself labelled with the dummy action, "#", as in Section 2, thus turning $s$ into a SCBC. Let $h$ be an abstraction homomorphism and let $\mathcal{A}_{h(L)}$ be the (minimal) LTS representing $h(L)$. Let $s_0, \overline{s}_0$ be the initial states of $\mathcal{A}_L$ and $\mathcal{A}_{h(L)}$ respectively.

**Definition A.1.** A state $s$ in $\mathcal{A}_L$ is said to *contribute* to a state $\overline{s}$ in $\mathcal{A}_{h(L)}$ if and only if there exists $w \in L$ such that $s_0 \xrightarrow{w} s$ and $\overline{s}_0 \xrightarrow{h(w)} \overline{s}$.

When constructing $\mathcal{A}_{h(L)}$, each state $\overline{s}$ in $\mathcal{A}_{h(L)}$ is automatically put into correspondence with a subset of states in $\mathcal{A}_L$; this subset is precisely the set of states that contributes to $\overline{s}$, so the 'contribution' relation can be effectively inferred.

As mentioned, our result takes an existing theorem and refines it. The theorem is as follows:

**Theorem A.2** (Ochsenschläger (1992)). *The homomorphism $h$ is WCC on $L$ if and only if for all pairs $(s, \overline{s}) \subseteq \mathcal{A}_L \times \mathcal{A}_{h(L)}$ such that $s$ contributes to $\overline{s}$,*

$$\exists v \in cont(\overline{s}, h(L)) \text{ such that } cont(v, h(cont(s, L))) = cont(v, cont(\overline{s}, h(L)))$$

We present some new definitions that allow us to re-cast this Theorem A.2 in more useful terms.

**Definition A.3.** Let $s$ be a state in $\mathcal{A}_L$, $\overline{s}$ a state in $\mathcal{A}_{h(L)}$ to which $s$ contributes. Then we say that $s$ is *WCC with* $\overline{s}$ if and only if $\exists v \in cont(\overline{s}, h(L))$ such that

$$cont(v, h(cont(s, L))) = cont(v, cont(\overline{s}, h(L)))$$

**Definition A.4.** Let $s$ be a state in $\mathcal{A}_L$. Then $s$ is a *WCC-state* if and only if, for all $\overline{s}$ to which $s$ contributes, $s$ is WCC with $\overline{s}$.

In light of this definition, Theorem A.2 may be re-phrased as in Theorem A.5.

**Theorem A.5.** *The homomorphism $h$ is WCC on $L$ if and only if, for all states $s$ in $\mathcal{A}_L$, $s$ is a WCC-state.*

The results of Theorems A.2 and A.5 suggest then that we must decide, for every single state in $\mathcal{A}_L$, whether the state is a WCC-state. Our refinement comes from the observation that it is in fact sufficient just to pick a representative state from each strongly-connected bottom component (SCBC) of $\mathcal{A}_L$, chosen arbitrarily, and check whether each of these states are WCC-states. In all fairness, we should stress here that if a system had more than one SCBC, few homomorphisms would be WCC on it.

**Theorem A.6.** *Let $C_1, C_2, ..., C_N$ be the SCBCs of $\mathcal{A}_L$, and let $s_1, s_2, ..., s_N$ be states such that $s_N$ is in $C_N$ for each $i = 1, 2, ..., N$. Then all states in $\mathcal{A}_L$ are WCC-states if and only if each $s_i$ is a WCC-state, for each $i = 1, 2, ..., N$.*

*Proof.* ($\Rightarrow$) Trivially true.

($\Leftarrow$) Let $s$ be any state in $\mathcal{A}_L$, and let $\overline{s}$ be any state in $\mathcal{A}_{h(L)}$ to which $s$ contributes; then let $w \in L$ be such that $s_0 \xrightarrow{w} s$ and $\overline{s}_0 \xrightarrow{h(w)} \overline{s}$. By Definition A.4, it suffices to prove that $\exists v \in cont(\overline{s}, h(L))$ such that

$$cont(v, h(cont(s, L))) = cont(v, cont(\overline{s}, h(L)))$$

From $s$, we can reach a strongly connected bottom component $C_i$[1], and so by assumption we can reach a WCC-state, $s_i$. Let $y$ be a word such that $s \xrightarrow{y} s_i$, and let $\overline{s}_i$ be the (unique) state such that $\overline{s} \xrightarrow{h(y)} \overline{s}_i$.

Now, $s_0 \xrightarrow{wy} s_i$ and $\overline{s}_0 \xrightarrow{h(w)h(y)=h(wy)} \overline{s}_i$ , so $s_i$ contributes to $\overline{s}_i$ and so, by definition of $s_i$,

$$\exists v' \in cont(\overline{s}_i, h(L)) : cont(v', h(cont(s_i, L))) = cont(v', cont(\overline{s}_i, h(L))) \qquad \text{(A.1)}$$

Firstly, we will prove that

$$cont(v', h(cont(s_i, L)) \subseteq cont(h(y)v', h(cont(s, L))) \qquad \text{(A.2)}$$

as follows: let $x \in cont(v', h(cont(s_i, L))$. Then $v'x \in h(cont(s_i, L))$. Therefore, there is $z \in cont(s_i, L)$ such that $h(z) = v'x$. Since $s \xrightarrow{y} s_i$, we see that $yz \in cont(s, L)$. Therefore, $h(y)h(z) \in h(cont(s, L))$; i.e. $h(y)v'x \in h(cont(s, L))$. Therefore, $x \in cont(h(y)v', h(cont(s, L))$, hence (A.2).

Therefore, combining (A.1) and (A.2), we see that

$$cont(v', cont(\overline{s}_i, h(L))) \subseteq cont(h(y)v', h(cont(s, L))) \qquad \text{(A.3)}$$

Since $\overline{s} \xrightarrow{h(y)} \overline{s}_i$, we have that

$$cont(v', cont(\overline{s}_i, h(L))) = cont(h(y)v', cont(\overline{s}, h(L)))$$

Plugging this into (A.3) we get

$$cont(h(y)v', cont(\overline{s}, h(L))) \subseteq cont(h(y)v', h(cont(s, L)))$$

Since the inclusion ($\supseteq$) always holds, we finally have that

$$cont(h(y)v', cont(\overline{s}, h(L))) = cont(h(y)v', h(cont(s, L)))$$

Let $v = h(y)v'$ $[\in cont(\overline{s}, h(L))]$; then for all pairs $(s, \overline{s}) \subseteq \mathcal{A}_L \times \mathcal{A}_{h(L)}$ such that $s$

[1]In general, for any automaton, each state in the automaton can always reach either a SCBC or a deadlock state. Since $\mathcal{A}_L$ is assumed to contain no deadlocks, the latter is not an option.

contributes to $\overline{s}$, there exists $v \in cont(\overline{s}, h(L))$ such that

$$cont(v, h(cont(s, L))) = cont(v, cont(\overline{s}, h(L)))$$

So $s$ is a WCC-state. Hence result. □

This is the key theorem in this Appendix. Combining it with Theorem A.5, we see that deciding whether $h$ is WCC on $L$ reduces to deciding whether, for a pair $s \in \mathcal{A}_L$, $\overline{s} \in \mathcal{A}_{h(L)}$, we have that:

$$\exists v \in cont(\overline{s}, h(L)), cont(v, h(cont(s, L))) = cont(v, cont(\overline{s}, h(L)))$$

i.e. whether $s$ is WCC with $\overline{s}$. The remainder of this section is geared towards defining concepts and proving results that will allow us to decide this condition, thus filling the blanks left in [Ochsenschläger (1992)].

**Definition A.7.** Let $\mathcal{A}_{h(cont(s,L))}$ be the minimal automaton representing the language $h(cont(s, L))$, $\mathcal{A}_{cont(\overline{s},h(L))}$ be the minimal automaton representing the language $cont(\overline{s}, h(L))$. Let $S_0$ and $\overline{S}_0$ represent the initial states in $\mathcal{A}_{h(cont(s,L))}$ and $\mathcal{A}_{cont(\overline{s},h(L))}$ respectively.

Note that the sets $cont(v, h(cont(s, L))$ and $cont(v, cont(\overline{s}, h(L))$ correspond to states in the automata $\mathcal{A}_{h(cont(s,L))}$ and $\mathcal{A}_{cont(\overline{s},h(L))}$ respectively, namely the states reached in each automaton by reading $v$; that is,

$$cont(v, h(cont(s, L))) = cont(succ(S_0, v), h(cont(s, L)) \tag{A.4}$$

and

$$cont(v, cont(\overline{s}, h(L)) = cont(succ(\overline{S}_0, v), cont(\overline{s}, h(L))) \tag{A.5}$$

This observation leads directly to Lemma A.8.

**Lemma A.8.** *The state $s$ in $\mathcal{A}_L$ is WCC with the state $\overline{s}$ in $\mathcal{A}_{h(L)}$ if and only if there is a word $v \in cont(\overline{s}, h(L))$ such that*

$$cont(succ(S_0, v), h(cont(s, L))) = cont(succ(\overline{S}_0, v), cont(\overline{s}, h(L)))$$

*Proof.* Proof follows immediately from Definition A.3, combined with (A.4) and (A.5). □

**Definition A.9.** Let $(S, \overline{S})$ be a state in $\mathcal{A}_{h(cont(s,L))} \times \mathcal{A}_{cont(\overline{s},h(L))}$. If it is not the case that $\forall a \in \Sigma$, $a$ is enabled at $S$ in $\mathcal{A}_{h(cont(s,L))}$ if and only if $a$ is enabled at $\overline{S}$ in $\mathcal{A}_{cont(\overline{s},h(L))}$, then we call $(S, \overline{S})$ a *bad state.*

The next theorem is the main Theorem of this Appendix, and leads to an algorithm for deciding WCC of $h$ on $L$, as we will describe once the theorem has been proved.

**Theorem A.10.** *Let $s$ be a state in $\mathcal{A}_L$, $\overline{s}$ a state in $\mathcal{A}_{h(L)}$ to which $s$ contributes. Then $s$ is a WCC state with $\overline{s}$ if and only if there is a state in $\mathcal{A}_{h(cont(s,L))} \times \mathcal{A}_{cont(\overline{s},h(L))}$ that cannot reach a* bad state *in $\mathcal{A}_{h(cont(s,L))} \times \mathcal{A}_{cont(\overline{s},h(L))}$.*

*Proof.* ($\Rightarrow$) Assume that $s$ is WCC with $\overline{s}$, and so let $v \in cont(\overline{s}, h(L))$ be such that $cont(succ(S_0, v), h(cont(s, L)) = cont(succ(\overline{S}_0, v), cont(\overline{s}, h(L))$. For brevity, we will write $S = succ(S_0, v)$, $\overline{S} = succ(\overline{S}_0, v)$.

The state $(S, \overline{S})$ is in $\mathcal{A}_{h(cont(s,L))} \times \mathcal{A}_{cont(\overline{s},h(L))}$ [as $(succ(S_0, v), succ(\overline{S}_0, v))$ clearly is]. We claim that $(S, \overline{S})$ cannot reach a *bad state*; proof is by contradiction.

So assume that $(S, \overline{S})$ *can* reach a *bad state* which we will denote $(S', \overline{S}')$; let $w$ be such that $(S, \overline{S}) \xrightarrow{w} (S', \overline{S}')$. By definition of a bad state, there exists $a \in \Sigma$ such that either

$a$ is enabled at $S'$ in $\mathcal{A}_{h(cont(s,L))}$ but not at $\overline{S}'$ in $\mathcal{A}_{cont(\overline{s},h(L))}$

[in which case $wa \in cont(S, h(cont(s, L))$ but $wa \notin cont(\overline{S}, cont(\overline{s}, h(L)))$]; or

$a$ is not enabled at $S'$ in $\mathcal{A}_{h(cont(s,L))}$ but is enabled at $\overline{S}'$ in $\mathcal{A}_{cont(\overline{s},h(L))}$

[in which case $wa \notin cont(S, h(cont(s, L))$ but $wa \in cont(\overline{S}, cont(\overline{s}, h(L)))$].

In either case, we do not have that $wa \in cont(S, h(cont(s, L))$ if and only if $wa \in cont(\overline{S}, cont(\overline{s}, h(L)))$; thus $cont(S, h(cont(s, L)) \neq cont(\overline{S}, cont(\overline{s}, h(L))$. Therefore, from Lemma A.8, $s$ is not WCC with $\overline{s}$; a contradiction.

Thus $(S, \overline{S}) \in \mathcal{A}_{h(cont(s,L))} \times \mathcal{A}_{cont(\overline{s},h(L))}$ cannot reach a *bad state*; hence result.

($\Leftarrow$) Let $(S, \overline{S}) \in \mathcal{A}_{h(cont(s,L))} \times \mathcal{A}_{cont(\overline{s},h(L))}$ be a state that cannot reach a *bad state*. Since $(S, \overline{S}) \in \mathcal{A}_{h(cont(s,L))} \times \mathcal{A}_{cont(\overline{s},h(L))}$, we must have (from rules of construction of product automata) that there exists $v \in h(cont(s, L)) \cap cont(\overline{s}, h(L))$ such that $S = succ(S_0, v)$ & $\overline{S} = succ(\overline{S}_0, v)$.

In light of Lemma A.8, it suffices then to prove that

$$cont(S, h(cont(s, L)) = cont(\overline{S}, cont(\overline{s}, h(L))$$

i.e. that

$$w \in cont(S, h(cont(s, L)) \Leftrightarrow w \in cont(\overline{S}, cont(\overline{s}, h(L))$$

Again, proof is by contradiction. Assume not; then there exists $w$ such that either

i) $w \in cont(S, h(cont(s, L))$ but $w \notin cont(\overline{S}, cont(\overline{s}, h(L))$; or

ii) $w \notin cont(S, h(cont(s, L))$ but $w \in cont(\overline{S}, cont(\overline{s}, h(L))$

Let $w'$ be the largest prefix of $w$ such that $w'$ is in both of $cont(S, h(cont(s, L))$ and $cont(\overline{S}, cont(\overline{s}, h(L))$ [$w' = \varepsilon$ satisfies this requirement, so such a $w'$ does indeed exist]. Let $a$ be the (unique) action such that $w'a \preceq w$. Let $(S', \overline{S}') = succ((S, \overline{S}), w')$ [so $S' = succ(S, w')$ & $\overline{S}' = succ(\overline{S}, w')$]. Now, $(S', \overline{S}')$ is reachable from $(S, \overline{S})$, which is not marked as 'may reach a bad state', so $(S', \overline{S}')$ is not 'bad'; i.e. $a$ is enabled at $S'$ in $\mathcal{A}_{h(cont(s,L))}$ if and only if $a$ is enabled at $\overline{S}'$ in $\mathcal{A}_{cont(\overline{s},h(L))}$.

If case (i) holds, then $a$ is enabled at $S'$ in $\mathcal{A}_{h(cont(s,L))}$ [since $w \in cont(S, h(cont(s, L))$ implies that $w'a \in cont(S, h(cont(s, L)))$], but not at $\overline{S}'$ in $\mathcal{A}_{cont(\overline{s},h(L))}$ [else we would have $w'a \in cont(\overline{S}, cont(\overline{s}, h(L))$ and $w'a \in cont(S, h(cont(s, L))$, contradicting the definition of $w'$], which contradicts the fact that $(S', \overline{S}')$ is not 'bad'.

If case (ii) holds, we have a similar contradiction; hence result. □

## A.3 The Algorithm, and Commentary

The last theorem, Theorem A.10, strongly hints at an algorithm for deciding WCC of $h$ on $L$, which we will now detail.

Firstly, we must find all SCBCs of $\mathcal{A}_L$; label them $C_1, C_2, ..., C_N$, and pick a state $s_i$ (completely arbitrarily) from $C_i$ for each $i = 1, 2, ..., N$. All of this can be accomplished in $O(|\mathcal{A}_L|)$ [Aho et al. (1974)].

From Theorem A.6, it suffices to show that each $s_i$ is a WCC-state, so we need only show how to decide whether a given state $s$ in $\mathcal{A}_L$ is a WCC-state.

To check that a given state $s$ in $\mathcal{A}_L$ is a WCC-state, we first construct the automaton $\mathcal{A}_{h(cont(s,L))}$; to do this, we temporarily reset the initial state of $\mathcal{A}_L$ to be $s$, and determinise the result. This is the most expensive operation of the whole procedure; formally, it is $O\left(2^{|\mathcal{A}_L|}\right)$, but since the abstracted version of a system is usually much, much smaller than the original (after all, this is why we construct $\mathcal{A}_{h(L)}$ instead of just using $\mathcal{A}_L$!), one imagines that it will in fact be much smaller than this; very probably, of comparable size to $Aut_{h(L)}$. Then we must find all states $\overline{s}$ in $\mathcal{A}_h(L)$ to which $s$ contributes; as mentioned after Definition A.1, this can be accomplished very easily, directly after the construction of $\mathcal{A}_h(L)$. The number of such $\overline{s}$ if of course bounded by $\mathcal{A}_h(L)$.

For each such $\overline{s}$, we must check whether $s$ is WCC with $\overline{s}$. To do this, we must first compute the automaton $\mathcal{A}_{cont(\overline{s},h(L))}$. This operation can be accomplished instantly, as it merely involves temporarily setting the initial state of $\mathcal{A}_h(L)$ to $\overline{s}$, and so we have that $\left|\mathcal{A}_{cont(\overline{s},h(L))}\right| \leq \left|\mathcal{A}_{h(L)}\right|$. We must then compute the product of this automaton with $\mathcal{A}_{h(cont(s,L))}$ to give the automaton $\mathcal{A}_{h(cont(s,L))} \times \mathcal{A}_{cont(\overline{s},h(L))}$; this can be performed in $O\left(\left|\mathcal{A}_{h(cont(s,L))}\right| \left|\mathcal{A}_{cont(\overline{s},h(L))}\right|\right)$, which from above is less than or equal to $O\left(\left|\mathcal{A}_{h(cont(s,L))}\right| \left|\mathcal{A}_{h(L)}\right|\right)$.

We must then find all *bad states* of the automaton $\mathcal{A}_{h(cont(s,L))} \times \mathcal{A}_{cont(\overline{s},h(L))}$, and then find all states in this automaton that may reach a *bad state*; both of these stages are linear in the size of the product automaton $\mathcal{A}_{h(cont(s,L))} \times \mathcal{A}_{cont(\overline{s},h(L))}$, and so can be performed in $O\left(\left|\mathcal{A}_{h(cont(s,L))}\right| \left|\mathcal{A}_{h(L)}\right|\right)$. If there are states in $\mathcal{A}_{h(cont(s,L))} \times \mathcal{A}_{cont(\overline{s},h(L))}$ that cannot reach a *bad state*, then $s$ is WCC with $\overline{s}$; else, it is not, and so $s$ is not a WCC-state, and so, by Theorem A.6, $h$ is not WCC on $L$. Thus, if we ever encounter an $\mathcal{A}_{h(cont(s,L))} \times \mathcal{A}_{cont(\overline{s},h(L))}$ where all states may reach a bad state, we may immediately terminate the algorithm with the value "false".

The algorithm scores over the "old" one (which was never actually articulated) by reducing the number of states $s$ in $\mathcal{A}_L$ that must be checked to see if they are WCC states from *all* states in $\mathcal{A}_L$ to just $N$, where $N$ is the number of SCBCs of $\mathcal{A}_L$. In general, $N$ is very much smaller than $|\mathcal{A}_L|$, so this represents a very significant saving.

## A.4 Summary

We have presented an algorithm for deciding whether an abstraction $h$ is weakly-continuation closed on a language $L$ for use in the case where the efficiently checkable sufficient condition (i.e. that the automaton representing $L$, $\mathcal{A}_L$, is not strongly-connected) doesn't hold. The algorithm is not technically efficient since it involves the determinisation of sub-automata of $\mathcal{A}_L$, but is more practical than the most obvious algorithm as it requires checks on only $N$ states of $\mathcal{A}_L$ (where $N$ is the number of strongly-connected bottom components of $\mathcal{A}_L$) rather than all states of $\mathcal{A}_L$.

# Appendix B

# An Improved Partial-Order Technique for Computing Abstractions

## B.1 Overview

This section presents a version of a persistent set selective search optimized specifically for finding *just* the abstraction of a language. It differs from Algorithm 3.1 in that it lifts the original proviso to which all persistent sets are subject and replaces it with what could be called a "weaker" one.[1] The result is relegated to the Appendices as, although the proviso is weaker in the sense described in Footnote 1, it may be that there is no practical means for exploiting it; indeed, as we will see when we present the algorithm that incorporates the proviso (Algorithm B.1), the fact that we must re-visit a state and expand the number of actions followed from it until the proviso is satisfied may well actually make it *less* efficient than that of Algorithm 3.1. Also, the algorithm very probably does not lead to a trace reduction, as does Algorithm 3.1 .

The algorithm presented in this section was directly inspired by the necessity of efficiently computing the abstraction of a system behaviour in order for the result of Chapter 5 to be of any use.

## B.2 The Original Proviso

Recall that the original proviso that a persistent set at the state $s$ had to satisfy was that it had to lead to a state not in the current stack, or else be trivial (i.e. the full set of

[1]It is weaker in the sense that using the proviso as in Algorithm 3.1 automatically fulfills this "new" proviso for all states.

actions). We briefly explore why it was used, in order to contrast it with the weakened proviso used in the result in this Appendix.

Consider the following (rather silly) situation, as shown in Figure B.1.

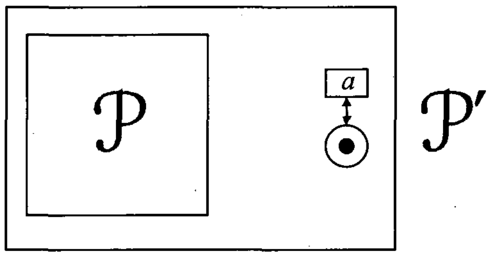


FIGURE B.1: A rather large system specification.

The Petri net $\mathcal{P}$ represents some colossal system specification, perhaps that of a global communications system specified down to the atomic level. This system has been expanded to form a new system, $\mathcal{P}'$, by means of introducing an additional action, $a$, with a hidden image under the abstraction $h$, and a single place connected to $a$. This action does not interact in any way with $\mathcal{P}$, and so is completely ($h$-compatibly) independent of *all* actions in $\mathcal{P}$. In light of this, and the fact that it is enabled at the initial state of $\mathcal{P}'$, we see that $a$ is persistent at the initial state. Thus, if we ignored the proviso imposed on candidate persistent sets, a perfectly valid partial-order reduction $R$ of the monster specification $\mathcal{P}'$ could be as shown in Figure B.2:

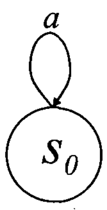


FIGURE B.2: A partial-order reduction of Figure B.1.

Now, one really cannot argue that the behaviour of $\mathcal{P}'$ is adequately represented in this reduction[2] due to the huge amount of actions 'excluded' from the reduction. A rather less silly and more realistic situation occurs when the set of actions followed at each successor of a state in the persistent set selective search is confined to some proper subset $A \subset \Sigma$ of the actions of the system. In this situation, the construction of the automaton will likely terminate rapidly and prematurely, with none of the actions in $\Sigma - A$ being represented at all in the resulting automaton. It is just this sort of situation, where whole 'important' sequences of actions are not represented in the reduction, which the proviso is designed to prevent; when all actions followed from a state all lead to "old" states (those in the stack), it is an indication that the construction might terminate "prematurely" without due consideration of other actions, and so we either try to follow more actions that lead

[2] We told you it was a silly example.

to "new" states, or we just follow all actions enabled, just to be sure. In other words, it is a "fairness" condition on the set of actions followed.

## B.3 Statement & Proof of the Result

We show that a variant of the persistent set selective search, with a different proviso than that used in the trace reduction algorithm (Algorithm 3.1), leads to an automaton with the same abstract image as the whole language. The proof of this is difficult (perhaps even impossible) for the usual definition of persistent sets (Definition 2.23), so we adopt a more restrictive definition, which was briefly introduced in Section 4.3.1 as a more workable definition. We define "strong persistent sets" as follows:

**Definition B.1.** Let $L \subseteq \Sigma^*$ be a language over $\Sigma$, $\Delta \subseteq \Sigma \times \Sigma \times L$ an independence relation over $\Sigma$, and $\Delta_{weak} \subseteq \Sigma \times \Sigma$ the weakened version of this relation (see Definition 4.16).

Let $w \in L$, $P \subseteq En(w)$ a non-empty subset of the set of enabled actions at $w$. Then $P$ is *strongly-persistent at* $w$ if and only if for all $p \in P$, and all actions $a$ appearing in words in $cont(w, L) \cap (\Sigma - P)^*$, $(p, a) \in \Delta_{weak}$.

If the weak independence relation used is $h$-compatible, then we say that $P$ is an $h$-compatible strongly persistent set at $w$.

**Definition B.2.** Let $\Delta^h$ be an $h$-compatible independence relation (see Definition 3.2). Then the language $R_L^{\Delta^h}$ is said to be an *h-compatible persistent reduction of a language* $L$ if and only if the following hold:

i) $R_L^{\Delta^h} \subseteq L$;

ii) For all $w' \in R_L^{\Delta^h}$, the set of actions $cont(w', R_L^{\Delta^h}) \cap \Sigma$ followed is $h$-compatibly strongly persistent at $w'$ in $L$; and

iii) For all $w' \in R_L^{\Delta^h}$; $h(cont(w', L)) \neq \varepsilon \implies h(cont(w', R_L^{\Delta^h})) \neq \varepsilon$.

The final condition means that at each state $s$ in the automaton representing $R_L^{\Delta^h}$, if the continuation from $s$ in $L$ contains any non-hidden actions, then the continuation from $s$ in the automaton representing $R_L^{\Delta^h}$ must also contain a non-hidden action. This is the "new proviso" that each state $s$ must fulfil. Unlike the original proviso, which is decidable for $s$ without having to explore from $s$, this one may require us to explore from $s$ multiple times, each time exploring a larger sub-automaton of the original $\mathcal{A}_L$; see Lines 21–38.

We claim that the language $R_L^{\Delta^h}$ allows us to compute the abstraction of $L$ under $h$; more specifically, that $h(R_L^{\Delta^h}) = h(L)$. First, though, we need a few preliminary Lemmas. For the next few Lemmas, we let $L \subseteq \Sigma^*$ be a language over $\Sigma$, $\Delta \subseteq \Sigma \times \Sigma \times L$ be an independence relation and $\Delta_{weak}$ be the weakened version of $\Delta$.

**Lemma B.3.** *Let $w \in L$, $v = xaby \in cont(w, L)$ with $(a, b) \in \Delta_{weak}$ and $a, b \in cont(w, L)$. Then $v' = xbay$ is in $cont(w, L)$ and $cont(wv, L) = cont(wv', L)$.*

*Proof.* By definition of $\Delta_{weak}$, and from the assumption that $a, b \in cont(w, L)$, we have that $ba \in cont(wx, L)$ and that $cont(wxab, L) = cont(wxba, L)$. Since $y \in cont(wxab, L)$ and $cont(wxab, L) = cont(wxba, L)$, we have that $xbay = v' \in cont(w, L)$. That $cont(wv, L) = cont(wv', L)$ is an immediate consequence of the fact that $cont(wxab, L) = cont(wxba, L)$. □

**Lemma B.4.** *Let $w \in L$. If $b, a_1a_2...a_n \in cont(w, L)$, with $(b, a_i) \in \Delta_{weak} \forall 1 \leq i \leq n$, then*

$$a_1a_2...a_jb \in cont(w, L)$$

*for each $0 \leq j \leq n$.*

*Proof.* Assume otherwise; let $j$ be the smallest index such that $a_1a_2...a_jb \notin cont(w, L)$. Then $j \neq 0$, since $b \in cont(w, L)$.

By choice of $j$, $a_1a_2...a_{j-1}b \in cont(w, L)$, so $b$ and $a_j$ are in $cont(wa_1a_2...a_j, L)$ [the latter since $a_1a_2...a_{j-1}a_j \in cont(w, L)$ and using prefix-closure of $L$]. Therefore, $b \in cont(wa_1a_2...a_j, L)$, by Definition 4.14i). Thus $a_1a_2...a_jb \in cont(w, L)$, a contradiction. Hence result. □

**Lemma B.5.** *Let $w \in L$. If $b, a_1a_2...a_n \in cont(w, L)$, with $(b, a_i) \in \Delta_{weak} \forall 1 \leq i \leq n$. Define $v_j = a_1a_2...a_jba_{j+1}a_{j+2}...a_n$ for each $0 \leq j \leq n$. Then for each $0 \leq j \leq n$:*

*i) $v_j \in cont(w, L)$ and $cont(wv_j, L) = cont(wv_n, L)$; and*

*ii) if $\Delta_{weak}$ is $h$-compatible, then $h(v_j) = h(v_n)$.*

*Proof.* To prove i): assume otherwise; then let $j$ be the largest index such that

$$v_j \in cont(w, L) \text{ and } cont(wv_j, L) = cont(wv_n, L)$$

Note that $v_n \in cont(w, L)$ by Lemma B.4, so $j \neq n$.

By choice of $j$,

$$v_{j+1} \in cont(w, L) \text{ and } cont(wv_{j+1}) = cont(wv_n, L) \qquad \text{(B.1)}$$

Now $v_j$ is formed from $v_{j+1}$ by transposing the pair of actions $(b, a_{j+1})$ that are adjacent in $v_{j+1}$; since $(b, a_{j+1}) \in \Delta_{weak}$, and $b, a_{j+1} \in cont(wa_1a_2...a_j, L)$ [from assumption and Lemma B.4], we have, from Lemma B.3, that $v_j \in cont(w, L)$ and

$$cont(wv_j, L) = cont(wv_{j+1}, L) = cont(wv_n, L)$$

a contradiction. Hence result.

The proof of ii) is trivial; simply note that since $v_j$ is formed from $v_{j+1}$ by transposing the pair of actions $(b, a_{j+1}) \in \Delta_{weak}$ that are adjacent in $v_{j+1}$ and $\Delta_{weak}$ is $h$-compatible, we have that $h(v_j) = h(v_{j+1})$. A simple proof by contradiction very much along the lines of that used in i) is sufficient. This proof is omitted.

Hence result. □

**Lemma B.6.** *Let $w \in L$. Let $x = a_1a_2...a_n \in cont(w, L)$, $y = b_1b_2...b_n \in cont(w, L)$, with $(a_i, b_j) \in \Delta_{weak}$ and $yx \in cont(w, L)$. Then $xy \in cont(w, L)$ and $cont(wxy, L) = cont(wyx, L)$.*

*Proof.* Proof is by induction on $n = |x|$.

Case $n = 1$ [$x = a_1$].

Immediate from Lemma B.5.

Assume true for $n - 1$.

So $b_1b_2...b_ma_1a_2...a_{n-1}a_n \in cont(w, L)$ from the assumption that $yx \in cont(w, L)$. Therefore $b_1b_2...b_ma_1a_2...a_{n-1} \in cont(w, L)$ by prefix-closure of $L$. By induction hypothesis, then, we have

$$a_1a_2...a_{n-1}b_1b_2...b_m \in cont(w, L) \tag{B.2}$$

and

$$cont(wa_1a_2...a_{n-1}b_1b_2...b_m, L) = cont(wb_1b_2...b_ma_1a_2...a_{n-1}, L) \tag{B.3}$$

From (B.3) and the fact that

$$a_n \in cont(wb_1b_2...b_ma_1a_2...a_{n-1}, L)$$

we have that $a_n \in cont(wa_1a_2...a_{n-1}b_1b_2...b_m)$ and that

$$cont(wa_1a_2...a_{n-1}b_1b_2...b_ma_n, L) = cont(wb_1b_2...b_ma_1a_2...a_{n-1}a_n, L) = cont(yx, L) \tag{B.4}$$

From Lemma B.5, we have that

$$a_1a_2...a_{n-1}a_nb_1b_2...b_m \in cont(w, L) \tag{B.5}$$

(i.e. $xy \in cont(w, L)$) and

$$cont(xy, L) = cont(wa_1a_2...a_{n-1}a_nb_1b_2...b_m, L) = cont(wa_1a_2...a_{n-1}a_nb_1b_2...b_ma_n, L) \tag{B.6}$$

Chaining equations (B.6) and (B.4) gives us that $cont(wxy, L) = cont(yx, L)$. Hence result.

□

**Lemma B.7.** *Let $w \in L$. Let $x = a_1a_2...a_n \in cont(w, L)$, $y = b_1b_2...b_m \in cont(w, L)$, with $(a_i, b_j) \in \Delta_{weak}$. Then $yx \in cont(w, L)$.*

*Proof.* Induction on $n = |x|$ again.

Case $n = 1$ $[x = a_1]$.

Immediate from Lemma B.5.

Assume true for $n - 1$. Then

$$b_1b_2...b_ma_1a_2...a_{n-1} \in cont(w, L)$$

by induction hypothesis. From Lemma B.6:

$$a_1a_2...a_{n-1}b_1b_2...b_m \in cont(w, L) \tag{B.7}$$

and

$$cont(wa_1a_2...a_{n-1}b_1b_2...b_m, L) = cont(wb_1b_2...b_ma_1a_2...a_{n-1}, L) \tag{B.8}$$

Since $a_n \in cont(wa_1a_2...a_{n-1}, L)$ by hypothesis, as does $b_1b_2...b_m$, we see from the earlier Lemma B.5 that

$$a_nb_1b_2...b_m \in cont(wa_1a_2...a_{n-1}, L)$$

i.e.

$$a_1a_2...a_{n-1}a_nb_1b_2...b_m \in cont(w, L)$$

Hence result, by induction. □

**Theorem B.8.** *Let $R_L^{\Delta^h}$ be an h-compatible persistent reduction of L. Then $\forall w' \in R_L^{\Delta^h}$*

$$h(cont(w', R_L^{\Delta^h})) = h(cont(w', L))$$

*Proof.* The inclusion from left to right follows almost immediately from Definition B.2); see Lemma 3.9 for details. Thus it suffices to prove inclusion from right to left. Proof is by induction on the length of words in continuations in $L$.

We take as our induction hypothesis the statement

*If* $|v| = n$, *then* $\forall w' \in R_L^{\Delta^h}$ *such that* $v \in cont(w', L)$, *there is a* $v' \in cont(w', R_L^{\Delta^h})$ *such that* $h(v') = h(v)$.

Let $w' \in R_L^{\Delta^h}$ and $v = a_1a_2...a_n$ be any word in $cont(w', L)$. The base case $n = 0$ [$v = \varepsilon$] is immediate, since $v' = \varepsilon$ is in $cont(w', R_L^{\Delta^h})$ [since $w' \in R_L^{\Delta^h}$] and $h(v') = \varepsilon = h(v)$.

So assume true for all words of length less than $n$, and try to prove true for $n$.

The case where $h(v) = \varepsilon$ is again immediate [with $v' = \varepsilon$ in this case also], so we assume that $h(v) \neq \varepsilon$.

So $h(v) \in h(cont(w', L))$, so $h(cont(w', L)) \neq \varepsilon$, so by definition of $R_L^{\Delta^h}$, we must have that $h(cont(w', R_L^{\Delta^h})) \neq \varepsilon$. Therefore, either there is a non-hidden action [call it $p$] in $Pers(w')$, or there is some $x = x_1x_2...x_m \in cont(w', R_L^{\Delta^h})$ [$m \geq 1$] with $Pers(w'x_1x_2...x_i)$ for $0 \leq i < m$ not containing any non-hidden actions and $Pers(w'x)$ containing a non-hidden action, $p$. [For notational convenience, we set $x = \varepsilon$ in the first case]. Note that in either case, $h(x) = \varepsilon$.

Now:

Case i) There is no prefix $x' \in pre(x)$ such that such that any of the $a_i$'s is in $Pers(w'x')$.

We wish first to show that $xpv \in cont(w', L)$. This poses no problems in the case where $x = \varepsilon$, so we assume that $x = x_1x_2...x_m[m \geq 1]$.

None of the $a_i$'s is in $Pers(w'x_1x_2...x_j)$ for $j = 0...m-1$, but $x_{j+1}$ is. By definition of these persistent sets, then, $(a_i, x_j) \in \Delta^h_{weak}$ for all $1 \leq i \leq n, 1 \leq j \leq m$ [and by a similar argument, $(a_i, p) \in \Delta^h_{weak}$ for all $1 \leq i \leq n$]. Thus, for each pair of actions taken from $v \in cont(w', L)$ and $xp \in cont(w', L)$ respectively, that pair is in $\Delta^h_{weak}$. Therefore, from Lemma B.7,

$$xpv \in cont(w', L)$$

Since $(a_i, p) \in \Delta^h$ for all $1 \leq i \leq n$, and $h(p) \neq \varepsilon$ and $h(v) \neq \varepsilon$, we deduce that

$$h(v) = h(p)^k$$

for some $k \geq 1$.

Let $l$ be the largest index such that $h(a_l) = h(p)$; then

$$h(a_1a_2...a_{l-1}) = h(p)^{m-1}$$

Now $xpa_1a_2...a_{l-1} \in cont(w', L)$ by prefix-closure of $L$. Let

$$u = a_1a_2...a_{l-1} \in cont(w'xp, L)$$

[so $|u| < n$]. Then by induction hypothesis, there is $u \in cont(w'xp, L)$ with $h(u') = h(u)$.

Since $h(x) = \varepsilon$, we have that

$$h(xpu') = h(pu') = h(p)h(p)^{m-1} = h(p)^m = h(v)$$

So setting $v' = xpu' \in cont(w', R_L^{\Delta^h})$ does the trick.

Case ii) There is some prefix $x' \in pre(x)$ such that some of the $a_i$'s are in $Pers(w'x')$.

Let $x'$ be the smallest such prefix. We wish first to show that $x'v \in cont(w', L)$. If $x' = \varepsilon$ then this is no trouble. Otherwise, $x' = x_1x_2...x_l$ for some $1 \leq l \leq m$. Using similar reasoning to that in case i), we see that by minimality of $x'$, $(a_i, x_j) \in \Delta^h_{weak}$ for all $1 \leq i \leq n$,$1 \leq j \leq l$.

Let $j$ be the smallest prefix such that $a_j \in Pers(w'x')$. Then $(a_j, a_k) \in \Delta^h_{weak}$ for all $1 \leq k < j$. Since $a_j \in Pers(w'x')$, $a_j \in cont(w', R_L^{\Delta^h}) \subseteq cont(w', L)$, as does $a_1a_2...a_{j-1}$. Therefore, by Lemma B.5, $a_ja_1a_2...a_{j-1} \in cont(w', L)$ and

$$cont(w'a_ja_1a_2...a_{j-1}, L) = cont(w'a_1a_2...a_{j-1}a_j, L)$$

[so $a_ja_1a_2...a_{j-1}a_{j+1}...a_n \in cont(w', L)$] and $h(a_ja_1a_2...a_{j-1}) = h(a_1a_2...a_{j-1}a_j)$, so that

$$h(a_ja_1a_2...a_{j-1}a_{j+1}...a_n) = h(a_1a_2...a_{j-1}a_ja_{j+1}...a_n) = h(v) \tag{B.9}$$

Since $a_j \in Pers(w'x')$, we have that $w'x'a_j \in R_L^{\Delta^h}$.

Let $u' = a_1a_2...a_{j-1}a_{j+1}...a_n \in cont(w'x'a_j, L)$. Then by induction hypothesis, there is $u' \in cont(w'x'a_j, R_L^{\Delta^h})$ with $h(u') = h(u)$.

Now $h(xa_ju') = h(a_ju) = h(a_ja_1a_2...a_{j-1}a_{j+1}...a_n) = h(v)$ using $h(x) = \varepsilon$ and (B.9). So $v' = xa_ju' \in cont(w', R_L^{\Delta^h})$ will do.

So we have proved for $n$ in both cases. Hence result, by induction. □

**Theorem B.9.**

$$h(R_L^{\Delta^h}) = h(L)$$

*Proof.* Immediate corollary to the previous theorem, Theorem B.8. □

We give now an algorithm (Algorithm B.1) that will construct such an $h$-compatible persistent reduction $R_L^{\Delta^h}$. Note that neither the algorithm nor the proofs rely on the system being a 'component-based' system [with components $C_1$, $I$ and $C_2$] even though finding the abstraction [*interface-level behaviour*] of these special types of systems was the original motivation for this result. Thus, the algorithm is completely generic.

It should be noted, though, that since in the special 'component-based' case we set $h$ to be the identity on $\Sigma^v$ so we could preserve variable bindings, *all* interface-level actions are dependent on one another [or rather, no pairs of interface-level actions are allowed in an $h$-compatible weak-independence relation]; that is, there is no "action renaming" allowed, and so two actions can belong to an $h$-compatible weak-independence relation only if one of them is hidden; i.e. is in $C_1$ or $C_2$. This poses some special problems when implementing this algorithm, which we will briefly discuss later.

Each state $s$ visited in the construction has a Boolean flag $s.NonHiddenCont$ that is set to true if and only there is a non-hidden action in the continuation of $s$ in $R_L^{\Delta^h}$. It can be shown that if a state $s$ is such that $s.NonHiddenCont$ is still set to false upon termination of the algorithm, then $h(cont(s, L)) = \varepsilon$, so our construction does indeed produce a an $h$-compatible persistent reduction of $L$ as defined in Definition B.2.

If a state $s$ is still unmarked after exploring all actions in its set of actions just followed, $T(s)$, then we explore again from $s$, this time with a persistent set $T(s)$ that includes some additional actions enabled at $s$ that were not previously explored, so there is a kind of "iterative" aspect to the algorithm. If $T(s)$ were the full set of enabled actions at $s$, then we "admit defeat" and resign ourselves to the fact that we cannot produce a non-hidden action in the continuation of $s$.[3]

If a non-hidden action is explored from a state, then this and all "predecessor" states [all states in the current *stack*] have their *NonHiddenCont* flags set to true [so a depth-first rather than breadth-first exploration is more or less essential for this algorithm]. Likewise, if during our construction we re-visit a state with its *NonHiddenCont* obligations fulfilled, then all states in the current *stack* may have their *NonHiddenCont* flags set to true.

As we see it, the main problem with implementing such an algorithm is that, in general, a set of actions is much more likely to be persistent at a state if it is hidden, since there are no "$h$-compatibility" issues to worry about.[4] In the case where we are trying to extract the interface-level behaviour for use with our first compositional verification result, then this bias towards "hidden" persistent sets is very much stronger since, for reasons described above, $h$-compatibility is much more of an issue in this case. So what we will probably see during construction is a strong tendency towards consideration mainly of hidden persistent sets, which do nothing towards fulfilling the proviso obligations for

[3] A brief examination of the algorithm tells us that if a state is unmarked even after the algorithm terminates, then all of its successors are unmarked and at each we must have "admitted defeat" after exploring the full set of actions enabled at the state. Thus, we see that if $s$ is unmarked when the construction is finished, then we will in fact have constructed the automaton $cont(s, L)$; i.e. $cont(s, L) = cont(s, R_L^{\Delta^h})$. This is how we can justify the statement that if a state $s$ has no non-hidden actions in its continuation in $R_L^{\Delta^h}$, then $s$ has no non-hidden actions in its continuation in $L$. This is a possible source of inefficiency of the algorithm, but we should point out that most "realistic" systems would rarely contain a state with a completely hidden continuation unless that state were a dead-end state, which would pose no problem since our construction would stop at such a state anyway.

[4] With our new relaxation of the restrictions on persistent sets, and with so-called "wild-card" hidden actions that are always guaranteed to form a persistent set when taken on their own, one could argue that the restriction of $h$-compatibility now becomes the primary force for disallowing candidate persistent sets.

```
$Q \leftarrow \phi; \delta \leftarrow \phi; s_0.NonHiddenCont = false; stack \leftarrow \phi;$
push ($s_0$) onto $stack$;
sub DFS()
    $s = top(stack)$;
    if $s.NonHiddenCont = true$ then   // Reached a state with a non-
        for each $s' \in stack$          // hidden continuation, so
            $s.NonHiddenCont = true$;    // update the .NonHiddenCont
        next                             // flags of all states in the
    endif                                // current stack.
    if $s \in Q$ then
        pop ($s$) from $stack$;
        exit sub     /* This state has been dealt with already */
    endif
    $s.NonHiddenCont = false; Q \leftarrow Q \cup \{s\}$;    /* This is a newly
                                                   created state*/
    if $En(s) = \phi$ then          // The state $s$ is a deadlock
        $\delta \leftarrow \delta \cup (s, \#, s)$;        // state. Add a loop to itself marked
        pop ($s$) from $stack$;     // with the dummy label, "#", pop it
        exit sub;                   // off the stack, and exit sub .
    endif
    $T = \phi$;    /* Begin an iterative loop to try and get
                 a $NonHiddenCont$ for the current state
                 by following progressively larger subsets $T$ of $En(s)$. */
    do until $s.NonHiddenCont = true$ or $T = En(s)$
        $NewActions = PersistentSetIncludingT(s, T)$;
        $T = PersistentSetIncludingT(s, T)$;
        for each $a \in NewActions$
            if $h(a) \neq \varepsilon$ then          // A non-hidden action found;
                for each $\overline{s} \in stack$;  // update all states in stack
                    $\overline{s}.NonHiddentCont = true$;    // accordingly
                next
            endif
            let $s'$ be such that firing $a$ at $s$ leads to $s$;
            $\delta \leftarrow \delta \cup (s, a, s')$;
            push ($s'$) onto $stack$;
            call DFS();
        next
    loop
    pop ($s$) from $stack$;
end sub
```

**Algorithm B.1** – Algorithm for Computing $R_L^{\Delta^h_{weak}}$

each state, nor indeed towards furthering the construction of the desired interface-level behaviour. As we have seen, this can be a bad thing since the algorithm tends to "panic" and "fully explore" from a state in an attempt to ensure the proviso holds at that state. What we suspect will increase the magnitude of the reduction is a heuristic which occasionally "throws in" a non-hidden persistent set[5] in an attempt to fulfill the proviso for as many states as possible. Just when to throw in these non-hidden sets to "even up the mix" is tricky, and is a topic for future research; nevertheless, we remain confident that strategies exist that will augment the state-space reduction in the majority of 'realistic' cases.

## B.4 Summary

We have presented an algorithm that uses a persistent-set selective search to find the abstraction of a language by weakening the proviso used in Algorithm 3.1 to which all explored states are subject. The proviso now reads "if a state has a non-hidden continuation in the full state-space, then it must have a non-hidden continuation in the partial-order reduced state-space", and will likely lead to an automaton that does not have the properties of a trace reduction. This weakened proviso cannot be checked as easily as the old proviso, so the algorithm may be required to "re-explore" a previously explored state if the proviso is found not to hold at that state, this time following more transitions in an attempt to get the proviso to hold. Whether this approach will truly be more efficient at just finding the abstraction of a language than Algorithm 3.1 requires experimental justification.

[5] Again, because the problem of $h$-compatibility is greatly exacerbated in the component-based case, this persistent set will very likely be trivial.

# Appendix C

# Commuting Limits for Non-Regular Languages

## C.1 Overview

The result in this Appendix is a very small and obscure one. It is relegated to the appendices for both this reason and also because it answers in the negative a question that, even if answered in the affirmative, would really have been primarily of theoretical rather than practical interest. It is included mainly because it took a fair amount of effort to resolve, and for its value as a curiosity .

An open question from Dr Ultes-Nitsche's thesis is whether the *Commuting Limit Theorem* holds for WCC homomorphisms on non-regular, prefix closed languages. The result had already been proved for general homomorphisms on regular, prefix closed languages $L$, and it simply states that

$$h(\lim(L)) = \lim(h(L)) \tag{C.1}$$

The result is in fact sufficient to prove that "$\lim(L) \models_{WF} h^{-1}(\mathcal{P}) \Leftrightarrow \lim(h(L)) \models_{WF} \mathcal{P}$ if and only if $h$ is WCC on $L$". Note that the inclusion of the LHS in the RHS in (C.1) always holds, whatever the nature of $h$ and $L$. The reason why we wanted (C.1) to hold in the case of non-regular, prefix closed languages is that it could perhaps have opened a new avenue into model-checking infinite state-spaces using abstraction and weak continuation-closure, although as said before it is more likely that no real practical application would have been found.

For general $h$ defined on non-regular $L$, (C.1) does not hold; one counter-example arises from setting

$$L = pre\left(\{a^n b^n | n \in \mathbb{N}\}\right)$$

a prefix-closed, non-regular language, and defining $h(a) = \varepsilon$ and $h(b) = B$. Then

$\lim(L) = \{a^\omega\}$, so $h(\lim(L)) = \{\varepsilon\}$. But $B^\omega \in \lim(h(L))$, so (C.1) is violated. It had been noticed that in this and a few other such counter-examples, $h$ was highly non-WCC and it seemed reasonable that the commuting limit theorem held in the non-regular case when $h$ was WCC on $L$.

However, after many failed attempts to prove the result, a search was launched for a counter example, and relatively soon one was discovered. Let

$$L = pre\left(\{a^n b^n c(b \cup c)^* | n \in \mathbb{N}\}\right), h(a) = \varepsilon, h(b) = B \text{ and } h(c) = C$$

It is easy to verify that: $L$ is prefix-closed and non-regular; $h$ is WCC on $L$; and that $B^\omega \in \lim(h(L))$ but $B^\omega \notin h(\lim(L))$, so we have a counter-example. Moreover, $L$ is in that class of languages representable by deterministic stack automata, the most "fundamental" non-regular languages; thus we cannot 'retreat' and try to prove that the theorem holds for a more restricted class of non-regular languages. However, it is very easily shown (although we will not do so here) that the result of (C.1) does in fact hold if $h$ is "non-erasing"; i.e. hides no actions. This, though, is the least useful type of abstraction homomorphism.

It seems that the highly non-WCC nature of the original counter-examples arose not because of any conflict between the notions of WCC and non-commuting limit, but rather because of the preponderance of *maximal words* in the counter-examples. This statement is in need of some clarification, so note that if a language $L$ has a maximal word, $w$ say, and a word $w'$ with $h(w) = h(w')$ and $h(cont(w', L)) \neq \{\varepsilon\}$, then $h$ cannot be WCC on $L$. This is because $h(cont(w', L)) \neq \varepsilon$ implies that $cont(h(w'), h(L))[= cont(h(w), h(L))] \neq \{\varepsilon\}$. Now any non-trivial word $v \in cont(h(w), h(L))$ will give $cont(v, h(cont(w, L))) = \phi$ because $w$ is maximal, and so $h$ cannot be WCC on $L$.

In the original example above, there were infinitely many such pairs $w, w'$, namely $w = a^n b^n$, $w' = a^{n+1} b^n$ for each $n \in \mathbb{N}$, thus explaining the highly non-WCC nature of the counter-examples. It was the recognition of this problem that first led us to append the $(b \cup c)^*$ to every word in $L$, which in turn led us to the proper counter-example.

## C.2 Summary

It was conjectured that the Commuting Limit Theorem would hold for prefix-closed, non-regular languages as long as the abstracting homomorphism was weakly continuation-closed on the language. In fact, this turned out to be not the case, and this Appendix presents a counter-example to the conjecture.

# Appendix D

# Deciding Satisfaction Within Fairness

## D.1 Overview

This Appendix details an algorithm for deciding satisfaction within fairness of a property by a language. It is not presented as a major part of the thesis due to its triviality.

## D.2 Algorithm 1

Let $L \subseteq (\Sigma \cup \{\#\})^*$ be a language represented by a deterministic LTS, $\mathcal{A}_L$, and let $\mathcal{P} \subseteq (\Sigma \cup \{\#\})^\omega$ be a property represented by a Büchi-automaton $\mathcal{A}_\mathcal{P}$. Note that, when treated as a Büchi-automaton, the automaton $\mathcal{A}_L$ represents the language $\lim(L)$.

It can be shown [Nitsche and Ochsenschläger (1996)] that

$$\lim(L) \models_{WF} \mathcal{P} \Leftrightarrow pre(\lim(L)) \subseteq pre(\lim(L) \cap \mathcal{P}) \tag{D.1}$$

Since $L$ contains no maximal words (the dummy action "#" was introduced specifically to prevent this situation), we have that $pre(\lim(L)) = L$, so (D.1) becomes

$$\lim(L) \models_{WF} \mathcal{P} \Leftrightarrow L \subseteq pre(\lim(L) \cap \mathcal{P}) \tag{D.2}$$

The condition (D.2) can be written as

$$\lim(L) \models_{WF} \mathcal{P} \Leftrightarrow L \cap \overline{pre(\lim(L) \cap \mathcal{P})} = \phi \tag{D.3}$$

where $\overline{L'}$ for a language $L' \subseteq (\Sigma')^*$ denotes the complement of the language $L'$, $(\Sigma')^* - L'$. The condition $L \cap \overline{pre(\lim(L) \cap \mathcal{P})} = \phi$ is easily decided using elementary automaton algorithms; first, we form the Büchi-automaton representing the $\omega$-language $\lim(L) \cap \mathcal{P}$ by taking the Büchi-product of $\mathcal{A}_L$ and $\mathcal{A}_\mathcal{P}$ ; then, we find the automaton representing $pre(\lim(L) \cap \mathcal{P})$. This is done by repeatedly removing all deadlock states and SCBCs containing no accepting states until neither of these two operations can be performed, at which point we set all states to be *accepting.* We then complement this automaton (note that an automaton must be determinized before this operation can be performed) to give an automaton representing the language $\overline{pre(\lim(L) \cap \mathcal{P})}$. Finally, we take the product of this automaton and $\mathcal{A}_L$, and check whether this final automaton is empty, as per (D.3).

Efforts were made to try and improve the efficiency of this algorithm, with limited success: whilst the new algorithm was superficially less complex, it still involved the determinization of the automaton $pre(\lim(L) \cap \mathcal{P})$, by far the most (formally) expensive of the operations listed above. It was therefore, from the point of view of computational complexity, almost identical to the original algorithm, and this is why we are presenting it in an Appendix, rather than in the main text. We comment on the exact algorithms to use at each stage, and upon the computational complexity.

## D.3 Algorithm 2

Firstly, as with the first case, we must compute the Büchi-product of the automata $\mathcal{A}_L$ and $\mathcal{A}_\mathcal{P}$; ordinarily, this process is more complicated (and yields a larger final automaton) than that required to compute the product of two ordinary automata [Thomas (1990)], due to subtleties involving faithfulness to the acceptance conditions of both component automata. In this special case, however, this is not an issue due to the trivial acceptance conditions of automaton $\mathcal{A}_L$; in fact, it can be shown that the usual algorithm of Algorithm D.1 will suffice.

Each state in the resulting automaton, then, is a pair $(s_L, s_\mathcal{P})$ with $s_L \in \mathcal{A}_L$, $s_\mathcal{P} \in \mathcal{A}_\mathcal{P}$. We now have a Büchi-automaton representing $\lim(L) \cap \mathcal{P}$, which we will call $\mathcal{A}_{L\times\mathcal{P}}$. We now use this to construct the automaton representing $pre(\lim(L) \cap \mathcal{P})$ using the method described in the old algorithm, and call it $\mathcal{A}^{pre}_{L\times\mathcal{P}}$. We then determinise this and call the result $(\mathcal{A}^{pre}_{L\times\mathcal{P}})_{\text{det}}$. Note that any state $S \in (\mathcal{A}^{pre}_{L\times\mathcal{P}})_{\text{det}}$ may be written as $S = \{(s^1_L, s^1_\mathcal{P}), (s^2_L, s^2_\mathcal{P}), ..., (s^n_L, s^n_\mathcal{P})\}$ for some $n$, where again $s^i_L \in \mathcal{A}_L$, $s^i_\mathcal{P} \in \mathcal{A}_\mathcal{P}$ for $i = 1, 2, ..., n$. We will show that we may detect violations of $\lim(L) \models_{WF} \mathcal{P}$ during the process of determinising $\mathcal{A}^{pre}_{L\times\mathcal{P}}$, and that if no such violations are detected by the time we have fully constructed $(\mathcal{A}^{pre}_{L\times\mathcal{P}})_{\text{det}}$, then $\lim(L) \models_{WF} \mathcal{P}$.

First, we need a couple of lemmas; the lemmas are presented without proof as, although they are "obvious", the proofs are fairly complex and make use of the details of the determinization algorithm.

```
$Q \leftarrow \phi; NewStates \leftarrow (s_0, \overline{s}_0)$;    /* Unbarred states are assumed to be in
                                    $\mathcal{A}$; barred ones in $\mathcal{A}'$ */
do while $NewStates \neq \phi$
    pick $(s, \overline{s}) \in NewStates$;
    $NewStates \leftarrow NewStates - (s, \overline{s})$;
    $Q \leftarrow Q \cup (s, \overline{s})$;
    if $\overline{s}$ is accepting then        // Since $\mathcal{A}$ is assumed to be an LTS,
        $(s, \overline{s})$ is accepting;      // the acceptance conditions of the
    else                              // states of the product automaton
        $(s, \overline{s})$ is not accepting;  // depend solely on those of $\mathcal{A}'$.
    endif
    for each $a \in \Sigma$ such that $a \in En(s) \cap En(\overline{s})$
        for each $(s', \overline{s}')$ such that $s' = succ(s, a)$ & $\overline{s}' = succ(\overline{s}, a)$
            if $(s', \overline{s}') \in Q$
                $NewStates \leftarrow NewStates \cup \{(s', \overline{s}')\}$;
            endif
            $\delta \leftarrow \delta \cup ((s, \overline{s}), a, (s', \overline{s}'))$;
        next
    next
loop
```

**Algorithm D.1** – Algorithm for constructing the product of two automata, $\mathcal{A}$ and $\mathcal{A}'$. The automata are assumed to contain no transitions labelled with $\varepsilon$, and $\mathcal{A}$ is assumed to be an LTS.

**Lemma D.1.** *Let* $S \in \left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$*; we may write* $S = \left\{\left(s^1_L, s^1_{\mathcal{P}}\right), \left(s^2_L, s^2_{\mathcal{P}}\right), ..., \left(s^n_L, s^n_{\mathcal{P}}\right)\right\}$ *as mentioned earlier. Then*

$$s^1_L = s^2_L = ... = s^n_L.$$

*Proof.* Proof is immediate from the determinization algorithm coupled with the fact that $\mathcal{A}_L$ is deterministic. □

In light of Lemma D.1, we will re-write $S = \left\{\left(s^1_L, s^1_{\mathcal{P}}\right), \left(s^2_L, s^2_{\mathcal{P}}\right), ..., \left(s^n_L, s^n_{\mathcal{P}}\right)\right\}$ as $S = \left(s; \left\{s^1_p, s^2_p, ..., s^n_p\right\}\right)$, where $s = s^1_L = s^2_L = ... = s^n_L$.

**Lemma D.2.** *Let* $S = \left(s; \left\{s^1_p, s^2_p, ..., s^n_p\right\}\right)$ *be a state in* $S \in \left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$, $a \in (\Sigma \cup \{\#\})^*$ *an action enabled at* $S$ *in* $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$, $S' = \left(s'; \left\{s'^1_p, s'^2_p, ..., s'^{n'}_p\right\}\right)$ *be the (unique) successor of* $S$ *following* $a$. *Then* $s'$ *is the (unique) state in* $\mathcal{A}_L$ *such that* $s \xrightarrow{a} s'$.

*Proof.* Again, the result is an obvious consequence of the workings of the determinization algorithm together with the fact that $\mathcal{A}_L$ is deterministic. □

**Lemma D.3.** $\lim(L) \models_{WF} \mathcal{P}$ *if and only if for all* $w \in L$, *there is a run of* $w$ *on* $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$.

*Proof.* Immediate from (D.1), after noting that the acceptance conditions of the automata representing $L$ and $pre(\lim(L) \cap \mathcal{P})$ are trivial, so that they accept a word if and only if there is a run of that word on the automaton. $\square$

**Theorem D.4.** *There is a run of $w$ on $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$ for each $w \in L$ if and only if, for each state $S = \left(s; \left\{s^1_p, s^2_p, ..., s^n_p\right\}\right)$ in $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$, there is no action $a$ enabled at $s$ in $\mathcal{A}_L$ that is not followed at $S$ in $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$.*

*Proof.* ($\Rightarrow$)Assume that $\forall w \in L$, there is a run of $w$ on $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$. Let

$$S = \left(s; \left\{s^1_p, s^2_p, ..., s^n_p\right\}\right)$$

be any state in $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$; then from the workings of the determinization algorithm, there must be some word $w$ such that the (unique) run of $w$ on $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$ leads to $S$. Let $a$ be any action enabled at $s$ in $\mathcal{A}_L$. Then by assumption, there is a run of $wa$ on $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$. Thus, since the run of $w$ on $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$ leads to $S$, $a$ must be enabled at $S$ in $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$. So for every action $a$ enabled at $s$ in $\mathcal{A}_L$, $a$ is enabled at $S$ in $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$.

($\Leftarrow$)Assume that for all states $S = \left(s; \left\{s^1_p, s^2_p, ..., s^n_p\right\}\right)$ in $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$, all actions $a$ enabled at $s$ in $\mathcal{A}_L$ are enabled at $S$ in $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$. Let $w = a_1a_2...a_m \in L$; we must show that there is a run of $w$ on $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$. Proof is by induction on $m$.

The case $m = 1$ ($w = a_1$) is immediate on noting that $a_1$ is enabled at $s_0$ in $\mathcal{A}_L$, and so it is enabled at the initial state $S_0 = \left(s_0, \left\{s^1_p, s^2_p, ..., s^k_p\right\}\right)$ of $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$; so there is a run of $w = a_1$ on $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$.

Assume true for $m - 1$ and try to prove for $m$. Now, by inductive hypothesis there is a run of $w' = a_1a_2...a_{m-1} \in L$ on $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$; let $S = \left(s; \left\{s^1_p, s^2_p, ..., s^n_p\right\}\right)$ be the (unique) state reached by following this run. Since the (unique) run of $w' = a_1a_2...a_{m-1}$ on $\mathcal{A}_L$ leads to the state $s$, $a$ must be enabled at $s$ in $\mathcal{A}_L$ (as there is a run of $w$ on $\mathcal{A}_L$, by assumption). Therefore, by assumption, $a$ is enabled at $S$. Thus there is a run of $w$ on $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$. $\square$

Now for a brief commentary on the computational complexity, starting with the formation of the product automaton of $\mathcal{A}_L$ and $\mathcal{A}_\mathcal{P}$, $\mathcal{A}_{L\times\mathcal{P}}$; as we have seen, each state in this automaton is of the form $(s_L, s_\mathcal{P})$, with $s_L \in \mathcal{A}_L$, $s_\mathcal{P} \in \mathcal{A}_\mathcal{P}$, so the number of states is bounded by $|\mathcal{A}_L|\,|\mathcal{A}_\mathcal{P}|$, where $|\mathcal{A}|$ for an automaton $\mathcal{A}$ represents the number of states of $\mathcal{A}$. The automaton $\mathcal{A}^{pre}_{L\times\mathcal{P}}$ created at the next stage will be a sub-automaton of $\mathcal{A}_{L\times\mathcal{P}}$; therefore, $\left|\mathcal{A}^{pre}_{L\times\mathcal{P}}\right| \leq |\mathcal{A}_{L\times\mathcal{P}}| \leq |\mathcal{A}_L|\,|\mathcal{A}_\mathcal{P}|$. The final stage is to determinise this automaton (although if the language $L$ does *not* satisfy $\lim(L) \models_{WF} \mathcal{P}$, we may stop before this determinization is complete); each state in this automaton, $\left(\mathcal{A}^{pre}_{L\times\mathcal{P}}\right)_{\text{det}}$, is of the form $S = \left(s; \left\{s^1_p, s^2_p, ..., s^n_p\right\}\right)$; the number of possible states is therefore bounded by $|\mathcal{A}_L|\,2^{|\mathcal{A}_\mathcal{P}|}$. This is the largest of the values so far, and so the whole process is $O\left(|\mathcal{A}_L|\,2^{|\mathcal{A}_\mathcal{P}|}\right)$.

One minor optimization to this algorithm exists; we may remove from $\mathcal{A}_{\mathcal{P}}$, prior to the construction of $\mathcal{A}_{L\times\mathcal{P}}$, those states in $\mathcal{A}_{\mathcal{P}}$ that cannot reach an accepting state that is reachable from itself. Informally, this is because any pairs $(s, \overline{s})$ in $\mathcal{A}_{L\times\mathcal{P}}$ where $\overline{s}$ has the aforementioned property will not appear in $\mathcal{A}^{pre}_{L\times\mathcal{P}}$, which is where the final check is made. We are currently aware of no other optimizations.

## D.4 Summary

This Appendix contains an algorithm for deciding whether a language satisfies a property within fairness. The algorithm is a fairly obvious one, and so was not presented in the main body of the thesis.

# Appendix E

# Comparison of Notions of Fairness

## E.1 Overview

In this Appendix, we define weakly and strongly fair satisfaction, and explore the relationship between them and the definition of satisfaction within fairness.

## E.2 Definitions

Let $S$ be a system represented by a minimal, deterministic finite LTS $\mathcal{A} = (Q, \Sigma, \delta, q_0)$, which encodes a prefix-closed regular language $L \subseteq \Sigma$. We assume that $L$ contains no maximal words, as per the construction on page 8 (so $\mathcal{A}$ contains no deadlock states). Let $B = \lim(L)$ be the behaviour of $S$, and $\mathcal{P} \subseteq \Sigma^\omega$ be any property over $\Sigma$.

**Definition E.1.** Let $w = a_0a_1... \in \lim(L)$ be a computation of $S$, and let $q_0q_1q_2...$ be the unique sequence of states such that $(q_i, a_i, q_{i+1} \in \delta)$ for all $i = 0, 1, 2, ....$ Then $w$ is a *weakly fair* computation of $S$ if and only if the firing of no action which is continuously enabled is postponed indefinitely. More formally, $w$ is weakly fair if and only if for all $a$ such that there is some $n_a$ for which $a$ is enabled at all of the states $q_m$ $(m \geq n_a)$, $a$ appears infinitely many times in the sequence $a_0a_1....$

**Definition E.2.** Let $w = a_0a_1... \in \lim(L)$ be a computation of $S$, and let $q_0q_1q_2...$ be the unique sequence of states such that $(q_i, a_i, q_{i+1} \in \delta)$ for all $i = 0, 1, 2, ....$ Then $w$ is a *strongly fair* computation of $S$ if and only if for all $a$ such that $a$ is enabled at the state $q_i$ for infinitely many $i$, $a$ occurs infinitely many times in the sequence $a_0a_1....$

**Definition E.3.** The behaviour $B = \lim(L)$ of a system $S$ *satisfies the property* $\mathcal{P} \subseteq \Sigma^\omega$ *under weak fairness* if and only if for all weakly fair computations $w \in \lim(L)$, $w \in \mathcal{P}$.

**Definition E.4.** The behaviour $B = \lim(L)$ of a system $S$ *satisfies the property* $\mathcal{P} \subseteq \Sigma^\omega$ *under strong fairness* if and only if for all strongly fair computations $w \in \lim(L)$, $w \in \mathcal{P}$.

## E.3 Statement and Proof of the Result

We prove now that given an $L$ as described at the beginning of this Appendix, any finite word $w \in L$ (partial computation of $S$) may be extended to an computation in $S$ that is strongly fair. The intuition is that after following a path to one of the strongly-connected bottom components (see Definition 2.12) of $\mathcal{A}$ from the state reached by following $w$ from $q_0$, we may generate an infinite path that includes all actions that can be enabled, infinitely many times.

**Lemma E.5.** *Let $L$ be a prefix-closed regular language containing no maximal words, and let $w \in L$. Then $\exists x \in$ cont$(w, \lim(L))$ such that the computation $wx \in \lim(L)$ is strongly fair.*

*Proof.* Let $s$ be the (unique) state in $\mathcal{A}$ reached by following $w$ from $q_0$. Then from the comment after Definition 2.12, we know that from $s$ we may reach either a deadlock state or a strongly-connected bottom component of $\mathcal{A}$. Since $\mathcal{A}$ is assumed to contain no deadlock states, we may always reach the former; let $C$ be such a SCBC of $\mathcal{A}$ reachable from $s$. Label the states in $C$ as $s_1, s_2, ..., s_n$, and let $y$ be a word that leads from $s$ to the state $s_1$ in $\mathcal{A}$. Since, from a state in $C$, we may only reach other states in $C$, the set of actions that can be followed from any state in $C$ is equal to $A = \bigcup_{s_i \in C} d \cup En(s_i)$, where $En(s_i)$ is the set of actions enabled at the state $s_i$, as defined in Section 2.3.1.[1]

We then build an $\omega$-word $\in$ *cont*$(s_1, L)$ that includes every action in $a$ infinitely many times; we do this by going through each element $a \in A$ in turn, following a path that leads to a state where $a$ is enabled and following $a$ (we know this is possible from the definition of $A$ and the fact that $C$ is strongly-connected), then doing the same for the next element of $A$ until all $a \in A$ have been included, then repeating the process for all elements of $A$, indefinitely. The $\omega$-word $y$ is the limit of this sequence. Let $x = yz$.

Thus, the word $xz \in \lim(L)$ has the property that each of the actions $a \in A$ occurs infinitely many times in $xz$. Since the only actions that may occur infinitely often in $xz$ are those in $A$, $xz$ is strongly fair. □

**Lemma E.6.** *Let $w \in \lim(L)$ be a strongly fair computation; then $w$ is also weakly fair.*

*Proof.* We prove this result by its contrapositive. Let $w = a_0a_1... \in \lim(L)$ be a computation that is *not* weakly-fair, and $q_0q_1q_2...$ be the sequence of states in $Q$ as described

[1]In other words, for any word $x \in$ *cont*$(s_i, L)$, $x$ will be a word over $A^*$.

in Definition E.1; then $\exists a$ such that there is some $n_a$ for which $a$ is enabled at all of the states $q_m$ $(m \geq n_a)$, but $a$ occurs only *finitely* many times in $w$.

But then $a$ is enabled at infinitely many of the $q_i$'s but does not occur infinitely many times in $w$; thus $w$ is not strongly fair. Hence result. □

**Theorem E.7.**

i) If $B$ satisfies $\mathcal{P}$ linearly, then $B$ satisfies $\mathcal{P}$ under weak fairness;

ii) If $B$ satisfies $\mathcal{P}$ under weak fairness, then $B$ satisfies $\mathcal{P}$ under strongly fairness;

iii) If $B$ satisfies $\mathcal{P}$ under strong fairness, then $B$ satisfies $\mathcal{P}$ within fairness.

*Proof.* To prove i): If $B$ satisfies $\mathcal{P}$ linearly, then for all $w \in B$, $w \in P$. Thus, trivially, any weakly-fair computation of $B$ is also in $\mathcal{P}$.

To prove ii): Assume that $B$ satisfies $\mathcal{P}$ under weak fairness. Let $w$ be any strongly fair computation of $B$; then $w$ is weakly-fair by Lemma E.6; therefore, $w \in P$. Thus for all strongly fair computations $w$ of $B$, $w \in P$.

To prove iii): Assume that $B$ satisfies $\mathcal{P}$ under strong fairness. Let $w$ be any partial computation of $B$ i.e. $w$ is a word in $L$. Then from Lemma E.5, there exists $x \in cont(w, \lim(L))$ such that $wx$ is strongly fair. Since $B$ satisfies $\mathcal{P}$ under strong fairness, $wx \in \mathcal{P}$. Thus, for all $w \in L, \exists x \in cont(w, \lim(L))$ such that $wx \in \mathcal{P}$. □

## E.4 Summary

We have defined weakly and strongly fair satisfaction and have proven that, if $B$ is a system behaviour and $\mathcal{P}$ a property, then:

$B$ satisfies $\mathcal{P}$ linearly $\Rightarrow$ $B$ satisfies $\mathcal{P}$ under weak fairness $\Rightarrow$ $B$ satisfies $\mathcal{P}$ under strong fairness $\Rightarrow$ $B$ satisfies $\mathcal{P}$ within fairness.

Thus, satisfaction within fairness can be described as more fair or more lenient than strong fairness.

# Bibliography

Martín Abadi and Leslie Lamport. The existence of refinement mappings. SRC Report 29, DEC System Research Center, July 1988.

Martín Abadi and Leslie Lamport. Composing specifications. SRC Report 66, DEC System Research Center, October 1990.

A.V. Aho, J.E Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4): 181–185, October 1985.

Rajeev Alur and Thomas A. Henzinger. Local liveness for compositional modeling of fair reactive systems. In Pierre Wolper, editor, *Computer Aided Verification (CAV) '95*, volume 939 of *Lecture Notes in Computer Science*, pages 166–179. Springer, 1995.

H. Barringer and D. Giannakopoulou. Proof rules for automated compositional verification through learning. In *Proc. of the Second Workshop on Specification and Verification of Component-Based Systems*, pages 14–21, September 2003.

G. Bruns. A practical technique for process abstraction. *Lecture Notes in Computer Science*, 715:37–49, 1993.

J.R. Büchi. On a decision method in restricted second order arithmetic. *Proceedings on the International Congress on Logic, Methodology and Philosophy of Science 1960*, pages 1–11, 1962.

H. J. Burkhardt, P. Ochsenschläger, and R. Prinoth. Product nets - a formal description technique for cooperating systems. GMD-Studien 165, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, September 1989.

Søren Christensen and Laure Petrucci. Modular analysis of Petri nets. *The Computer Journal*, 43(3):224–242, ???? 2000.

E. M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, Albuquerque, 1992.

J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification. In *In Proc. of the 9th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346. Springer-Verlag, April 2003.

Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: An investigation of decomposition for assume-guarantee reasoning.

E. Cohen. Asynchronous progress. In A. McIver and C. Morgan, editors, *Programming Methodology*. Springer, 2002.

Patrick Cousot and Rahida Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2-3):103–179, 1992. ISSN 0743-1066.

Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997. ISSN 0164-0925.

L. de Alfaro and T. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

Claudio de la Riva and Javier Tuya. Automatic generation of assumptions for modular verification of software specifications. *J. Syst. Softw.*, 79(9):1324–1340, 2006. ISSN 0164-1212.

D.E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, 1993.

S. Ehmety and L. Paulson. Mechanizing compositional reasoning for concurrent systems: some lessons. In *Formal Aspects of Computing*, volume 17, pages 58–68, 2005.

S. Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, New York, 1974.

N. Francez. *Fairness*. Springer-Verlag, New York, first edition, 1986.

P. Godefroid. Using partial-orders to improve automatic verification methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer Aided Verification (CAV) '90*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185, New Brunswick, 1991. Springer Verlag.

P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem.* PhD thesis, Université de Liège, Liège, Belgium, 1995.

P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Formal Methods in System Design*, volume 2, pages 149–164, April 1993.

S. Graf and C. Loiseaux. Property preserving abstractions under parallel composition. *Lecture Notes in Computer Science*, 668:644–657, 1993.

Susanne Graf. Verification of a distributed cache memory by using abstractions. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 207–219, London, UK, 1994. Springer-Verlag. ISBN 3-540-58179-0.

M. Hack. Decidability questions for petri nets. Technical Report 161, MIT, 1976.

M.A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, Mass., first edition, 1978.

T.A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43:135–141, 1992.

G. Holzmann. *Design and validation of computer protocols.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. ISBN 0-13-539925-4.

J.E Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Mass., first edition, 1979.

J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 54–69, Liege, Belgium, 1995. Springer Verlag.

K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer Verlag, 1997.

Cliff B. Jones. Specification and design of (parallel) programs. In R. E. A. MASON, editor, *Proceedings of the IFIP 9th World Congress*, volume 83 of *Information Processing*, pages 321–332, 1983.

O. Kupferman and M. Y. Vardi. Module checking revisited. In *Proc. 9th International Computer Aided Verification Conference*, pages 36–47, 1997.

R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, Princeton, New Jersey, first edition, 1994.

D. Lehmann and M. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *POPL*, pages 133–138, 1981.

Wayne Liu. Interaction abstraction for compositional finite state systems. In *SPIN*, pages 148–162, 2000.

M. Chiodo, T.R. Shiple, A.L. Sangiovanni-Vincentelli, and R.K. Brayton. Automatic Compositional Minimization in CTL Model Checking. In *IEEE /ACM International*

*Conference on CAD*, pages 172–178, Santa Clara, California, 1992. IEEE Computer Society Press.

K. McMillan. Compositional methods and systems. In *Verification of Digital and Hybrid Systems*, pages 138–151. Springer-Verlag, 2000.

Kenneth L. McMillan. A compositional rule for hardware design refinement. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 1997. ISBN 3-540-63166-6.

J. Misra. A discipline of multiprogramming: Programming theory for distributed applications, 2001.

Wonhong Nam and Rajeev Alur. Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In *ATVA*, pages 170–185, 2006.

U. Nitsche. Propositional linear temporal logic and language homomorphisms. In A. Nerode and Y.V. Matiyasevich, editors, *Proceedings of the 3rd International Symposium On Logical Foundations of Computer Science*, pages 265–277, Saint Petersburg, Russia, 1994. Springer Verlag.

U. Nitsche. Application of formal verification and behaviour abstraction to the service interaction problem in intelligent networks. *Journal of Systems and Software*, 40(3): 227–248, March 1998.

U. Nitsche and P. Ochsenschläger. Approximately satisfied properties of systems and simple language homomorphisms. *Information Processing Letters*, 60:201–206, 1996.

U. Nitsche and P. Wolper. Relative liveness and behaviour abstraction (extended abstract). In *Proceedings of the 16th ACM Symposium On Principles of Distributed Computing (PODC'97)*, pages 45–52, Santa Barbara, CA, 1997.

P. Ochsenschläger. Verifikation kooperierender Systeme mittels schlichter Homomorphismen. Arbeitspapiere der GMD 688, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, Oktober 1992.

P. Ochsenschläger. Verification of cooperating systems by simple homomorphisms using the product net machine. In A. Desel, A. Oberweis, and W. Reisig, editors, *Workshop: Algorithmen und Wekzeuge fur Petrinetze*, pages 48–53. Humboldt Universität Berlin, 1994.

S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.

D. Peled. Combining partial order reduction with on-the-y model checking. In *Proceedings of computer aided verication*, volume 818, pages 377–390. Springer-Verlag, 1994.

A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

A. Pnueli. In transition from global to modular temporal reasoning about programs. pages 123–144, 1985.

W. Reisig. *An Introduction to Petri Nets.* Springer-Verlag, 1985.

S. Graf and C. Loiseaux. A Tool for Symbolic Program Verification and Abstraction. In T. Kropf, R. Kumar, and D. Schmid, editors, *GI/ITG Workshop Formale Methoden zum Entwurf korrekter Systeme*, pages 122–138, Bad Herrenalb, 1993. Universität Karlsruhe, Interner Bericht Nr. 10/93.

Natarajan Shankar. Lazy compositional verification. *Lecture Notes in Computer Science*, 1536:541–564, 1998.

S. St James and U. Ultes-Nitsche. Computing property-preserving behaviour abstractions from trace reductions. In *Proceedings of the 20th ACM Symposium On Principles of Distributed Computing (PODC'01)*, pages 235–238, Newport (R.I.), 2001.

S. St James and U. Ultes-Nitsche. An optimised partial-order approach to the verification of system components. *New Technologies for Information Systems - Proceedings of the 1st International Workshop on Validation and Verification of Enterprise Information Systems (VVEIS 2003)*, pages 69–81, April 2003.

W. Thomas. Automata on infinite objects. In Jan van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 133–191. Elsevier, 1990.

Wolfgang Thomas. Computation tree logic and regular omega-languages. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 690–713, London, UK, 1989. Springer-Verlag. ISBN 3-540-51080-X.

U. Ultes-Nitsche. Application of formal verification and behaviour abstraction to the service interaction problem in intelligent networks. *Journal of Systems and Software*, 40(3):227–248, 1998.

U. Ultes-Nitsche. A persistent-set approach to abstract state-space construction in verification. In *Proceedings of the 26th Seminar on Current Trends in Theory and Practice in Informatics (SOFSEM)*, volume 1725 of *Lecture Notes in Computer Science*, pages 463–70, New Brunswick, 1999.

U. Ultes-Nitsche and S. St James. Weakly-continuation closed abstractions can be defined on trace reductions. In *Proceedings of the International Workshop on Verification and Computational Logic (VCL'2000)*, 2000.

A. Valmari. A stubborn attack on state explosion. In E.M. Clarke and R.P. Kurshan, editors, *CAV'90 - Computer Aided Verification 1990*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165. Springer Verlag, 1991a.

A. Valmari. Stubborn sets for reduced state space generation. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer Verlag, 1991b.

H. van der Schoot and H. Ural. An improvement on partial order model checking with ample sets, 1996.

Kimmo Varpaaniemi. Efficient detection of deadlocks in Petri nets. Research Report A26, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, October 1993.

P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In E. Best, editor, *LNCS715, CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 233–246. Springer Verlag, 1991.