Abstract

This thesis expands the usage of partial order reduction methods in reducing the state space of large models in model checking. The work done can be divided into two parts. In the first part we introduce two new ample conditions that utilise strongly connected components in place of two existing ample conditions that use cycles. We use these new conditions to optimise existing partial order reduction verifiers and extend them to verify nonblocking properties. We also introduce two selection strategies for choosing ample event sets and an improved ample algorithm in order to improve the efficiency of ample set computation, and investigate how the various combinations of these suggested algorithmic improvements effect several models of varying size. The second part of the thesis introduces the concept of using partial order reduction techniques in combination with compositional verification techniques. We introduce a modified version of the silent continuation rule that makes use of the independence relationship from partial order reduction methods and include algorithms by which they may be implemented in a model verifier. All of the original concepts developed in this thesis are also proven correct.

Contents

Chapter 1

Introduction

Model verification of discrete event systems [5] typically involves determining certain properties of different models. In order to do this, the various states that a system may be in are examined. Since a system is often represented by several indivudual components, each with its own set of states that it may be in, to determine a state of the whole system we construct a synchronous composition [11, 16] out of all of the seperate components. This synchronous composition represent every possible combination of states that the different components may be in. For large systems with many different components or with many states in each component, the number of states in this synchronous composition grows exponentially, resulting in what is called a *state-space explosion*. Since we may wish to determine propeties in various large systems like this, it is helpful to use methods by with the state-space explosion problem may be mitigated.

Once such method is *partial order reduction*. This method aims to exploit certain structures that are present in a synchronous composition in order the identify and eliminate redundant states. To do this we use use a relationship between the events in the system known as *independence*. If we notice that two event are independent, then we know that the order in which these events occur if they are ever both able to be performed in the same state, does not matter. This gives rise to some states that do not add unique behaviour in the system and thus may be removed.

Another method is *compositional verification* [22]. This approach is based around making abstraction of the components using several different rules that identify states that are able to be merged together. The merging of the states yields state space reduction in the component automata, which when composed together two at a time, may have the process repeated on the resulting composition. Continuing to reduce the components and synchronise until the system is small enough to verify is the goal of this process.

This thesis will develop methods for both partial order reduction and compositional verification. In particular a partial order reduction implementation will be introduced to verify both nonblocking and controllability properties. We will also offer several optimisations to the partial order reduction controllability verifier offered in [17] along with some new conditions and proofs for the correctness of the process. A new abstraction rule for use in compositional verification which utilises concepts from partial order reduction is also offered, along with the necessary algorithms for development and proofs of correctness.

The report is organised as follows. Chapter 2 introduces definitions and terminology used throughout the report and offers background information to set give context so that the later discussions can be readily interpreted. Chapter 3 explains the research done in implementing partial order reduction model verifiers for nonblocking and controllability. It includes subsections for detailing the algorithms that were developed, proofs of the various conditions introduced, experimental results and conclusions. Chapter 4 explains the research done in developing a new abstraction rule for compositional verification with ideas from partial order reduction. It includes subsections for detailing the algorithms that were developed, proofs of the various conditions introduced and conclusions. A functional implementation of the research given in Chapter 4 has not yet been realised so the experimental results are excluded from that chapter. Chapter 5 then concludes the thesis.

Chapter 2

Preliminaries

This chapter introduces the concepts and mathematical notation used throughout this thesis. Any additional notation or concepts specific to particular areas will be introduced as required in later chapters.

2.1 Automata

An automaton [6] (singular form of automata), also known as a finite state machine (FSM), is a collection of states and the transitions between those states using events. The states represent a certain configuration that the system being modelled can be in. As events are performed, a transition occurs changing the current state of the automaton, thus changing the configuration.

Definition 2.1. An *automaton* A is defined as the tuple $\langle \Sigma, S, S^\circ, \rightarrow, Q \rangle$ where:

- Σ is the complete set of events of the automaton, also referred to as the alphabet
- S is the complete set of states of the automaton
- $S^\circ \subseteq S$ is the set of *initial states* of the automaton. These are the states that the automaton can be in before any transitions have occurred.

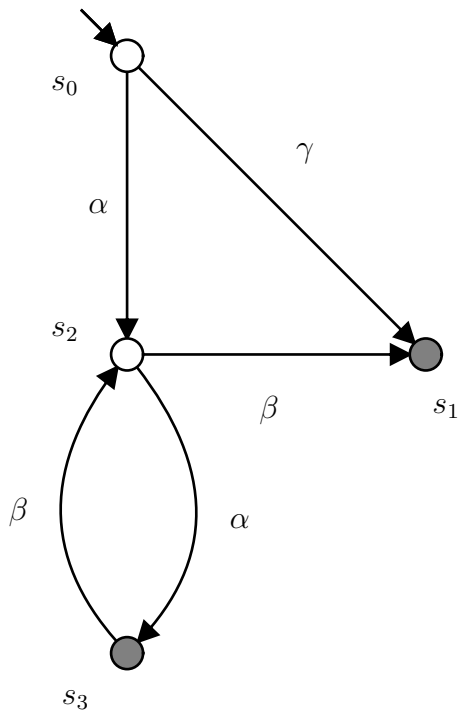

Figure 2.1: An example automaton.

- $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation of the automaton.
- $Q \subseteq S$ is the set of marked states.

An example automaton is given in figure 2.1. In this automaton we have $\Sigma = \{\alpha, \beta, \gamma\}$, $S = \{s_0, s_1, s_2, s_3\}$, $S^\circ = \{s_0\}$ and $Q = \{s_1, s_3\}$. We can also see that $(s_0, \alpha, s_2) \in \rightarrow$. This may also be expressed as $s_0 \xrightarrow{\alpha} s_2$. This represents the transition from state s_0 to state s_1 using event α.

Definition 2.2. Let $A = \langle \Sigma, S, S^\circ, \rightarrow \rangle$ be an automaton. An event $\alpha \in \Sigma$ is defined to be *enabled* in a state $s \in S$ if there exists a another state $s' \in S$ and a transition $s \xrightarrow{\alpha} s'$. This may also be represented as $s \xrightarrow{\alpha}$ if the target state of the transition is not identified. The set of events $enabled_A(s) \subseteq \Sigma$ is the set of all events that are enabled in s. If event β is not enabled from a state s, this is expressed as $s \not\xrightarrow{\beta}$.

In the automaton given in Figure 2.1 we can see that $\alpha, \beta \in \mathit{enabled}(s_2)$, and $\gamma \notin \mathit{enabled}(s_2)$. The following expressions then are all true: $s_2 \xrightarrow{\alpha}$, $s_2 \xrightarrow{\beta}$, $s_2 \not\xrightarrow{\gamma}$, $s_2 \xrightarrow{\beta} s_1$, $s_2 \xrightarrow{\alpha} s_3$.

Definition 2.3. A *path* is defined to be a sequence of transitions taken in an automaton. The length of a path π is expressed as $|\pi|$. The initial state of a path is the state from which the first transition occurs, and the end state of a path is the target state of the final transition in the path. Two paths may be composed together if the end state of the first path is the initial state of the second path. For two such paths π_0 and π_1 , this is expressed as $\pi_0 \circ \pi_1$.

In the automaton given in Figure 2.1 a valid path would be

$$s_0 \xrightarrow{\alpha} s_2 \xrightarrow{\alpha} s_1 \xrightarrow{\beta} s_2 \xrightarrow{\beta} s_3 \xrightarrow{\gamma} s_0$$

in which the initial state and the end state are both s_0. This may also be expressed as the composition of two paths such as

$$s_0 \xrightarrow{\alpha} s_2 \xrightarrow{\alpha} s_1 \circ s_1 \xrightarrow{\beta} s_2 \xrightarrow{\beta} s_3 \xrightarrow{\gamma} s_0$$

Definition 2.4. A *string* is a sequence of events from Σ written in succession such as $\sigma_1\sigma_2 \ldots \sigma_n$. The events of a path may be expressed as a string, for example the path $s \xrightarrow{\sigma_1} t \xrightarrow{\sigma_2} u$ would have the string $\sigma_1\sigma_2$. Strings may also be used in transitions to omit intermediate states, for example $s \xrightarrow{\sigma_1\sigma_2} u$. The set of all finite strings of events in Σ, including the empty string, is represented as Σ^* .

Definition 2.5. Let $A = \langle \Sigma, S, S^\circ, \rightarrow, Q \rangle$ be an automaton and $s, t \in S$. States can be considered *reachable* with respect to other states represented as $s \rightarrow t$, or with respect to automata represented as $A \rightarrow t$. In the latter case this means that state t is reachable from the initial state S°. In both instances there is an implied existence of a string $p \in \Sigma^*$ where $s \xrightarrow{p} t$, meaning that state t is reachable from state s using only the events from string p.

Definition 2.6. A *strongly connected component* is a maximal set of states $C \subseteq S$ with the property that for all states $s_i, s_j \in C$ we have $s_i \rightarrow s_j$.

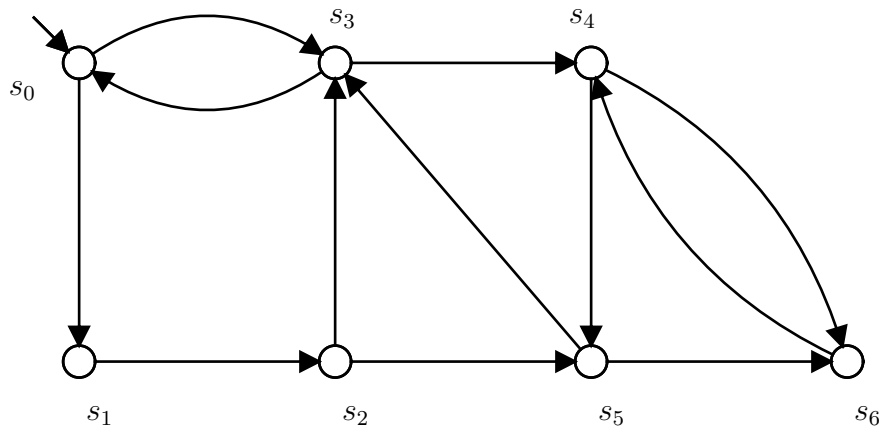

Figure 2.2: An automaton with one strongly connected component.

This means that every state in a strongly connected component is able to reach every other state in the strongly connected component. Figure 2.2 gives an example of an automaton with one strongly connected component. It can be seen that every state in this automaton is able to reach every other state. Figure 2.3 shows the same automaton as in Figure 2.2 but the transition from state s_5 to s_3 has been removed. As a result the automaton now has two strongly connected components $C_1 = \{s_0, s_1, s_2, s_3\}$ and $C_2 = \{s_4, s_5, s_6\}$.

Definition 2.7. An automaton is defined to be *deterministic* if it meets the following criteria:
Let $s_1, s_2, s_3 \in S$, $\alpha \in \Sigma$, then

$$s_1 \xrightarrow{\alpha} s_2 \wedge s_1 \xrightarrow{\alpha} s_3 \implies s_2 = s_3$$

and

$$s_1 \in S^\circ \wedge s_2 \in S^\circ \implies s_1 = s_2$$

Or more simply there cannot exist two transitions from a single state on the same event if those transitions would result in different states, also there maybe be only one initial state. Any automaton that is not deterministic is defined to be *non-deterministic*. An example of a non-deterministic automa-

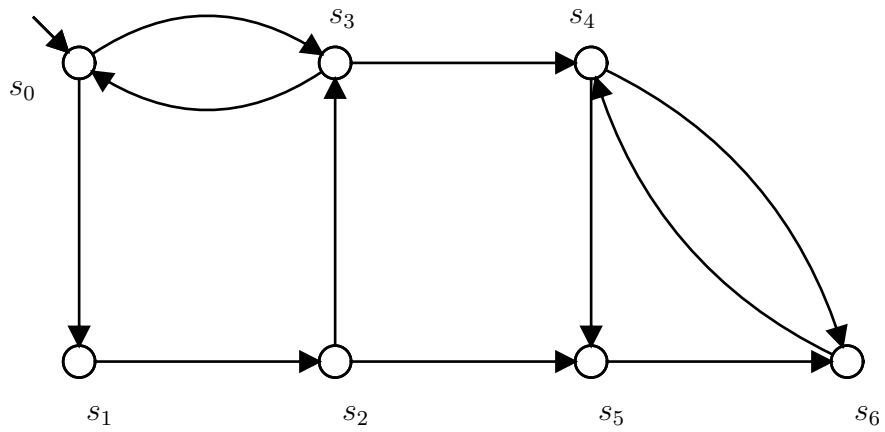

Figure 2.3: An automaton with two strongly connected components after removing a transition.

ton is given in Figure 2.4. This example automaton is non-deterministic due to the fact that there are two transitions from state s_0 using event α.

2.2 Synchronous Composition

A synchronous composition or synchronous product of several automata is achieved by taking several automata and synchronising them on their events. The result of this is a single large automaton that models the system as if each of the automata involved in the synchronous composition were executing their transition operations in parallel. This allows a large complicated system to be constructed by creating smaller, simpler component automata, each responsible for a part of the whole system and then constructing the synchronous composition from the components.

The idea behind this is that there will be a set of common events across the alphabets of each automaton and that the automata all run concurrently. An event α is only enabled in a state in the synchronous composition if every component automaton containing α in its alphabet has α enabled in its current state. When α is executed, each of these automata performs the

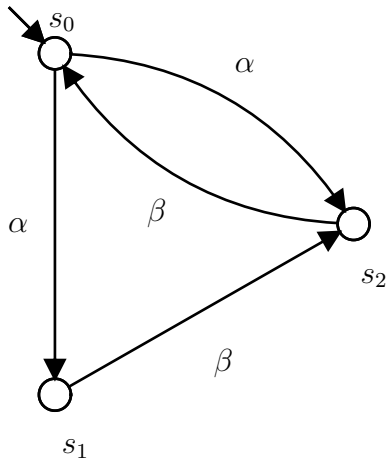

Figure 2.4: An example of a non-deterministic automaton.

appropriate transition and changes state, while all of the other automata remain unchanged. This is only possible however if each automaton that contains α in its alphabet has α enabled in its current state, otherwise α is disabled for the entire system. This effectively means that in an automaton that does not include α in its alphabet, a transition occurs on α in every state where the target state is the same as the current state for that automaton. This type of transition is referred to as a *selfloop*.

The synchronous composition then has states that each represent the combination of single states from each of the automata. The current state of the synchronous composition then translates into a current state in each of the automata, i.e., if the current state of the synchronous composition is its initial state, then the current state of every automaton involved in the synchronous composition will be the initial state also. When an event is executed from the current state of the synchronous composition this is analagous to each of the automata executing that event from its current state and the combination of all of the resulting states across all of the automata will be the target state of that transition in the synchronous composition. This process is called *lock-step synchronisation* or *handshaking*. Composing each of the

automata can be approached algorithmically and is defined mathematically as follows [4, 11]:

Definition 2.8. Let $M_1 = \langle \Sigma_1, S_1, S_1^\circ, \to_1 \rangle$ and $M_2 = \langle \Sigma_2, S_2, S_2^\circ, \to_2 \rangle$ be two automata. The *synchronous composition* of M_1 and M_2 is

$$M_1 || M_2 = \langle \Sigma_1 \cup \Sigma_2, S_1 \times S_2, S_1^\circ \times S_2^\circ, \to \rangle \text{ where}$$

- $(x, y) \xrightarrow{\alpha} (x', y')$ if $\alpha \in \Sigma_1 \cap \Sigma_2, x \xrightarrow{\alpha}_1 x'$ and $y \xrightarrow{\alpha}_2 y'$;
- $(x, y) \xrightarrow{\alpha} (x', y)$ if $\alpha \in \Sigma_1 \setminus \Sigma_2$ and $x \xrightarrow{\alpha}_1 x'$;
- $(x, y) \xrightarrow{\alpha} (x, y')$ if $\alpha \in \Sigma_2 \setminus \Sigma_1$ and $y \xrightarrow{\alpha}_2 y'$.

An example [16] of a small factory comprised of two machines and a buffer modelled using automata is given in Figure 2.5. The way that this system operates is that the two machines can either be idle, working, or broken; as respresented by the states I, W and B respectively. The subscript on each state represents the specific machine number ie. I_1 represents the initial state of machine 1. The buffer may either be empty or full; represented by states E and F respectively. Machine 1 works on one item at a time. Whenever machine 1 finishes working on an item then the item is placed in the buffer, causing the buffer to become full. Machine 2 may then start working by removing the item from the buffer, causing the buffer to become empty. When either machine is working it is possible for the machine to break, at which point any work that it was currently doing is lost and the machine must be repaired before it may start working again.

To model this behaviour, the two machines begin in the idle state and the buffer starts in the empty state. Since these automata are synchronised on their common events, only the s_1 event corresponding to machine 1 starting work is possible. This is because even though machine 2 would allow event s_2, the fact that the buffer has a transition involving s_2 but does not allow s_2 in state E means that the system is unable to perform event s_2, so machine 2 cannot start work. s_1 is allowed because even though machine 2 does have a

transition involving s_1 from its idle state, s_1 does not appear in the alphabet of machine 2, so there are implicit selfloops on every state in machine 2 using event s_1. In fact this is the case for every event from machine 1, so machine 2 will never prevent an event from machine 1 happening, and will always remain in the same state on execution of such an event. The reverse is also true so no events from machine 2 will affect the state of machine 1. After performing event s_1, machine 1 transitions to state W signifying that it is now working, whereas the buffer and machine 2 remain in states E and I respectively. From here events b_1 or f_1 are possible, representing machine 1 breaking down or finishing work. If event b_1 is taken then machine 1 tranistions to state B_1 and must receive an r_1 event before going back to state I_1, at which point the process can begin again. If event f_1 is taken then machine 1 returns to state I_1 and the buffer transitions to state F_1, signifying that machine 1 has placed an item into the buffer. From here event s_2 is now permitted as the buffer is no longer disabling it. Performing event s_2 takes the buffer back to state E while machine 2 transitions to state W_2. Now either f_2 or b_2 are available, representing machine 2 finishing work or breaking down as was the case with machine 1. Taking event b_2 works in the same way as b_1 did with machine 1, and taking event f_2 causes machine 2 to transition back to state I_2. Once this has happened the system is back in its initial state and these processes can begin again.

Figure 2.6 is the synchronous composition of the three component automata shown in Figure 2.5. Each of the states that exist in this diagram are given labels corresponding to the states that each of the component automata are in. For example the initial state is labelled $E.I.I$, which represents the states of the buffer, machine 1 and machine 2 in turn. This means that initially the buffer is in state E (empty), machine 1 is in state I (idle), and machine 2 is also in state I (idle). The subscripts are omitted here as the ordering can be used to determine which state belongs to which automaton. This example shows that even when considering a relatively small system where only a very small amount of possible operations can be performed, a fairly large and complicated synchronous composition is produced as it

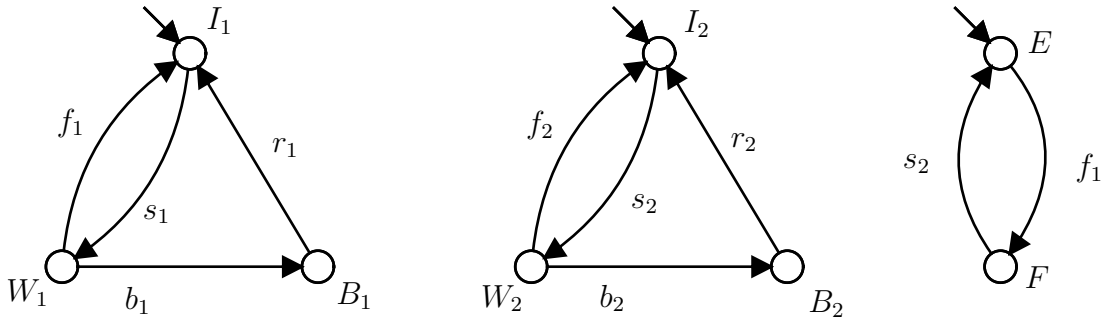

Figure 2.5: Small factory model.

mimics all of the possible interactions with the system.

Since there is this notion of local and global states in single automata and the synchronous composition respectively, notation is needed to define the relationship between them.

Definition 2.9. Let $M = M_0||M_1||\ldots||M_n$ be represented by the tuple $\langle \Sigma, S, S^\circ, \rightarrow \rangle$. A *global state* is then any state in S.

Definition 2.10. Let $s = (s_0, s_1, \ldots, s_n) \in S$ be a global state, where each s_i represents a local state of automaton M_i. $enabled_i(s) = enabled_{M_i}(s_i)$ represents the set of events that are enabled in the local state of M_i corresponding to global state s.

An issue does arise however when constructing the synchronous compositions. As the number of component automata grow along with the number of states and events in the component automata, what is seen is the number of states, or state space, of the synchronous composition increases exponentially in size. This phenomena is known as *state space explosion*. Eventually the state spaces of large models become too large for memory and so it is advantageous to apply techniques whereby the state space can be reduced in size while preserving the properties of interest of the model.

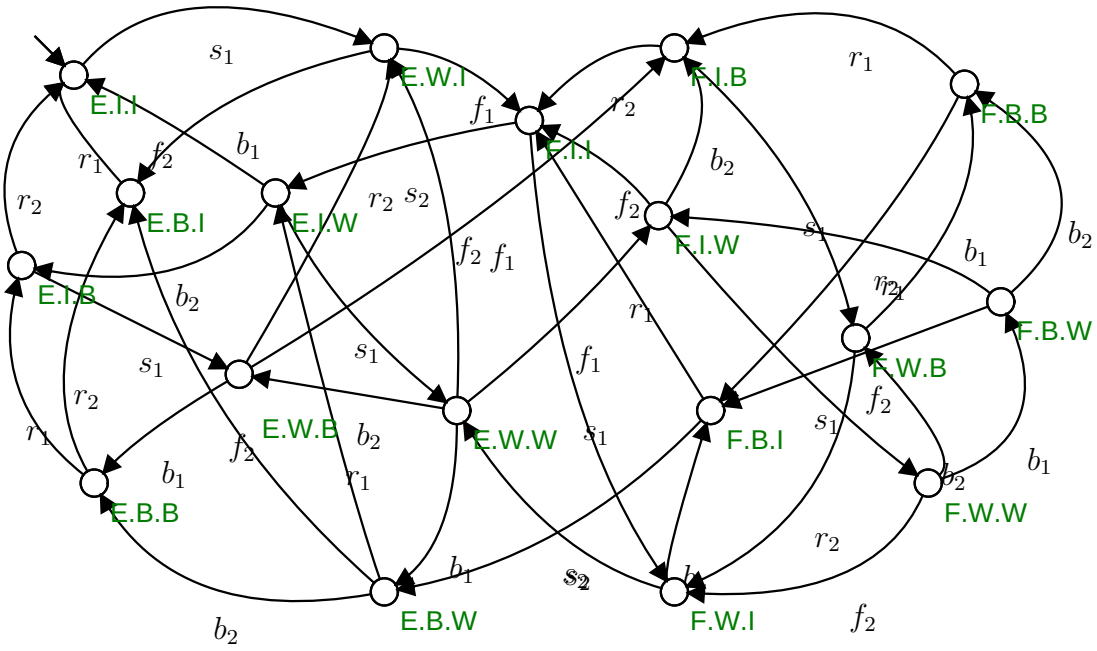

Figure 2.6: Synchronous composition of the small factory model.

2.3 Controllability Checking

The concept of controllability introduces the notion that there may exist bad states within a system. We call these bad states uncontrollable states. These states represent behaviour of the system that is undesirable, such as the doors to an elevator cabin being opened while the cabin is moving between floors for example. A system is said to be *controllable* if there are no states that are *uncontrollable*, otherwise the system is defined to be *uncontrollable*. These concepts are introduced in [16]. To determine the controllability of states several factors must first be observed.

- Events may be classified as either *controllable* or *uncontrollable*.
- Automata may be classified as either *specifications* or *plants*.

A *plant* automaton is an automaton that describes the behaviour of the physical system that is being modelled. A *specification* automaton is an

automaton that describes a controller for the physical system. An uncontrollable event is an event that will occur in the system outside of the control of the system itself or any user of the system. Events such as a car arriving at traffic lights or an incoming call would be examples of uncontrollable events in a systems modelling traffic lights and a system modelling a telecommunications centre respectively. A controllable event is an event that the system can perform in response to some other event, be it controllable or uncontrollable. In the same two example systems controllable events might be changing the lights from red to green or connecting an incoming call to an operator. Uncontrollable states occur whenever a specification automaton prevents the system from performing an uncontrollable event that the plants would otherwise allow. The reasoning behind this is that uncontrollable events can not be prevented from happening so any controllers must handle those events when they occur.

This can be described as follows:

Definition 2.11. Let G be a plant automaton and K be a specification automaton. K is *controllable* with respect to G if for every global state (s_G, s_K) that is reachable in $G||K$ and every uncontrollable event μ

$$\mu \in enabled_G(s_G) \implies \mu \in enabled_K(s_K)$$

Any global state for which this does not hold is an uncontrollable state. Any model in which all specification automata are controllable with respect to all plant automata is said to be a *controllable* model, otherwise the model is said to be uncontrollable. This is equivalent to saying the any model whose synchronous composition has no reachable uncontrollable states is controllable, otherwise it is uncontrollable.

Consider the synchronous composition given in Figure 2.7. This is the same model as Figure 2.6 except now we are considering the ideas of controllability covered above. In this model machine 1 and machine 2 are plants and the buffer is a specification. The controllable events are s_1, s_2, r_1 and r_2, and the uncontrollable events are b_1, b_2, f_1 and f_2. This makes sense as

a machine breaking or finishing the work it was doing are not events that can be controlled. What we see as a result of this are three uncontrollable states, $F.W.I$, $F.W.W$, and $F.W.B$, signified by the red crosses on each of them. Let us examine one of these states to determine why it is uncontrollable. The first of these states, $F.W.I$, has the buffer in state F, machine 1 in state W_1, and machine 2 in state I_2. If we refer to Figure 2.5 we can see that the enabled events in each of the component automata are s_2 in the buffer and machine 2, and f_1 in machine 1. The f_1 event has been declared uncontrollable however and is enabled in the plant automaton machine 1, so following the conditions for controllability above it must also be the case that each specification automaton enables f_1. This is not the case however, since the buffer disables f_1 in state F, leading to this state $F.W.I$ being an uncontrollable state. Since the buffer is uncontrollable with respect to machine 1 in this case, this means that this model is uncontrollable.

Definition 2.12. A *controllability counterexample* is a path taken from the initial state to an uncontrollable state in the synchronous composition.

For the uncontrollable state that was used in the previous example, a counterexample can be constructed by examining the diagram in Figure 2.7

$$E.I.I \xrightarrow{s_1} E.W.I \xrightarrow{f_1} F.I.I \xrightarrow{s_1} F.W.I$$

This counterexample represents a problem with the system, wherein the buffer which can only hold one item at a time has been put in a situation where it must accept a new item when it is already storing an item. This is known as buffer overflow. To address this problem, the buffer specification must be modified so that buffer overflow can not occur.

It is possible and is also useful to consider controllability of automata in another way. By replacing specification automata with plant automata that have transitions to states marked as a bad states every time a controllability problem is encountered, it can be shown that all controllability problems can be translated into reachability problems with respect to these new states. The

process for constructing these plant automata out of specification automata is as follows [8]:

Definition 2.13. Let $K = \langle \Sigma, S, \rightarrow, S^\circ, Q \rangle$ be a specification automaton and $\Sigma_u \subseteq \Sigma$ be the set of all uncontrollable events of K. The complete plant automaton $K^\perp$ for K is

$$K^\perp = \langle \Sigma, S \cup \{\perp\}, \rightarrow^\perp, S^\circ, Q \rangle$$

where $\perp \notin S$ is a new state and

$$\begin{aligned} \rightarrow^\perp = \rightarrow \cup \{\langle s, \alpha, \perp \rangle \mid s \in S, \alpha \in (\Sigma_u \cap \Sigma), s \overset{\alpha}{\not\rightarrow}\} \\ \cup \{\langle \perp, \beta, \perp \rangle \mid \beta \in \Sigma\} \end{aligned}$$

Using this translation the following is then true: K is controllable with respect to G if and only if there is no state $(x, \perp)$ with $x \in S_G$, reachable in $G||K^\perp$. In a synchronous composition involving more than one specification, the model is uncontrollable if there exists a reachable global state where at least one of the local states is $\perp$.

This interpretation in known as *plantification* and is useful in later chapters when proving the correctness of the partial order reduction method.

2.4 Conflict checking

Another property of systems that is often of interest is that of *conflict* or *blocking*. This idea deals with reachability within a system with regard to a set of states which are marked. These marked states often represent states in which the system may come to rest or terminate safely and as such an inability to reach these states from any given state can be undesirable. In diagrams, states that are shaded grey are the marked states. A system is said to be *nonblocking* if for all reachable states a path exists to a marked state, otherwise it is considered *blocking*. When several automata are involved then

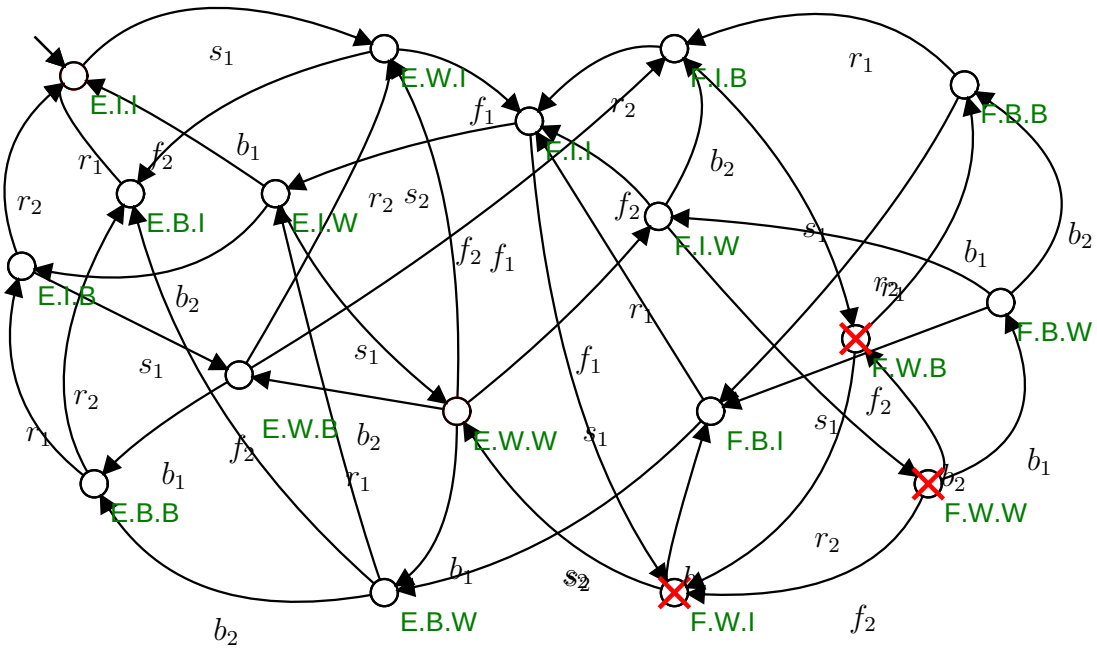

Figure 2.7: Synchronous composition of the small factory model with uncontrollable states.

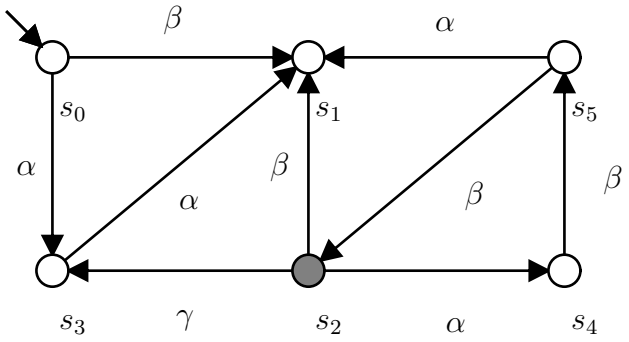

Figure 2.8: An automaton with blocking states.

the problem of blocking and nonblocking in the synchronous composition is often referred to as *conflicting* or *nonconflicting*.

Definition 2.14. An automaton $M = \langle \Sigma, S, S^\circ, \rightarrow, Q \rangle$ is nonblocking if for every state $s \in S$ where $M \rightarrow s$, we have $s \rightarrow t$ where $t \in Q$. Two automata M_1 and M_2 are nonconflicting if $M_1 || M_2$ is nonblocking.

There are two practical ways in which an automaton may be determined to be blocking. Either an unmarked state can have no outgoing transitions in which case clearly no marked states can be reached, or the only states reachable from an unmarked state are all unmarked and in turn cannot reach a marked state. In the former case this is referred to as *deadlock* and in the latter case this is referred to as *livelock*. Any states found to be in either deadlock or livelock are blocking states.

Consider the automaton in Figure 2.8. This automaton can be observed as being blocking. The only marked state is state s_2, and the only states that can reach it are the states s_4 and s_5. This means that states s_0, s_1 and s_3 are blocking states. It can be seen that each of the states s_0 and s_3 are in livelock as they each have outgoing transitions, yet no sequence of transitions from any of these states will ever reach s_2. State s_1 is in deadlock as it has no outgoing transitions and is itself unmarked. Adding a transition from s_1

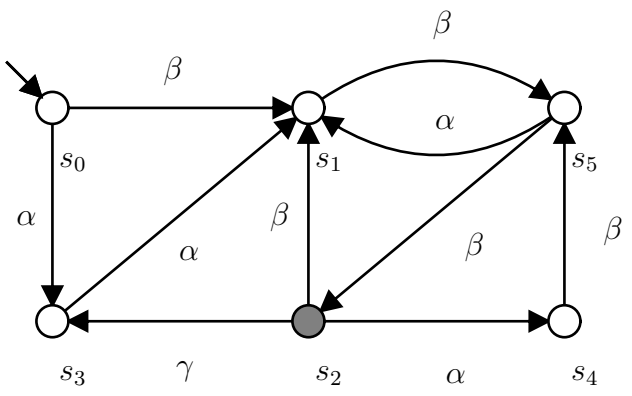

Figure 2.9: Automaton with blocking states resolved.

to either s_2, s_4 or s_5 would resolve the issue of blocking in this automaton, as pictured in Figure 2.9.

It will prove useful when constructing proofs that involve conflict verification to use an abstracted automaton model. For this abstraction we introduce a *termination event* ω and for each marked state, add a transition with ω to a special *dump state* $\perp$. The abstraction is defined as follows.

Definition 2.15. Let $M = \langle \Sigma, S, S^\circ, \rightarrow, Q \rangle$. The abstraction for M is then $M^\omega = \langle \Sigma, S \cup \{\perp\}, S^\circ, \rightarrow^\omega \rangle$ where $\perp \notin S$ is a new state and

$$\rightarrow^\omega = \rightarrow \cup \{\langle s, \omega, \perp \rangle, \langle \perp, \alpha, \perp \rangle \mid s \in Q, \alpha \in \Sigma\}$$

M^ω is then nonblocking for every state s where $M^\omega \rightarrow s$ we have $s \stackrel{p\omega}{\rightarrow}$ for some string $p \in \Sigma^*$

2.5 Partial Order Reduction

The ideas discussed in this section introduce concepts that are fundamental to the partial order reduction process outlined in this thesis. The two main concepts introduced are independence and ample sets. Independence is a relation between events and is used to identify structure in the state space

where there may be redundant states. The section on page 20 will describe the criteria by which events are considered to be independent and what this means in terms of an automaton. Ample sets are the result of the process by which reduction of the state space of an automaton is achieved. The ample set of a state s, denoted $ample(s)$, is a subset of $enabled(s)$. The section on page 21 will describe the criteria by which $ample(s)$ is able to be selected and shows how to apply those criteria with a small example.

2.5.1 Independence

The notion of *independence* [3] [6] [10] is an important one when considering how to reduce the state space of a synchronous composition. Since the reduction is made possible by eliminating redundancies in the full state space it is necessary to identify relationships between events that would give rise to such redundancies. This relationship is known as *independence*.

Definition 2.16. In an automaton $A = \langle \Sigma, S, S^\circ, \rightarrow \rangle$, two events $\alpha, \beta \in \Sigma$ are defined to be *independent* if in every state $s \in S$ where $\alpha, \beta \in enabled_A(s)$ they satisfy the following condition:

$$s \xrightarrow{\alpha} t_1 \text{ and } s \xrightarrow{\beta} t_2 \implies \text{there exists } v \in S \text{ where } t_1 \xrightarrow{\beta} v \text{ and } t_2 \xrightarrow{\alpha} v$$

Firstly this ensures that once α is performed from state s then the resulting state still has β enabled, and vice versa. This is equivalent to saying that it ensures that α does not *disable* β. Events that are not independent are defined to be *dependent*.

Definition 2.17. Let $A = \langle \Sigma, S, S^\circ, \rightarrow \rangle$ be an automaton and let $\alpha, \beta \in \Sigma, s_1, s_2 \in S$. α *disables* β in state s_1 if

$$\alpha, \beta \in enabled(s_1), s_1 \xrightarrow{\alpha} s_2 \text{ and } s_2 \not\xrightarrow{\beta}$$

Definition 2.16 also ensures that the order in which α and β are executed has no effect on the final resulting state. Another way to say this is to say that α and β *commute*. Figure 2.10 gives an example of an automaton where

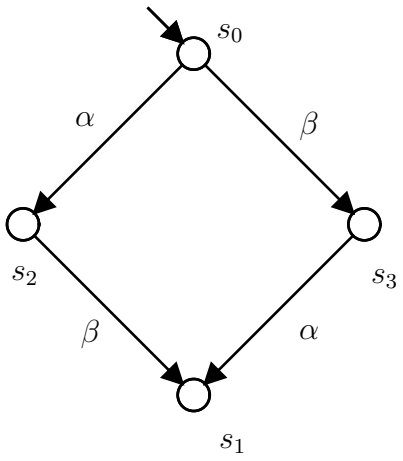

Figure 2.10: Automaton demonstrating independence of events α and β.

two events α and β are independent. It can be seen in this example that $enabled(s_0) = \{\alpha, \beta\}$. The target states of the transitions involving each of the events leave the other enabled, meaning that $s_0 \xrightarrow{\alpha} s_2$, $s_2 \xrightarrow{\beta}$ and $s_0 \xrightarrow{\beta} s_3$, $s_3 \xrightarrow{\alpha}$. It can also be seen that the targets of those transitions result in the same state, seen by observing the transitions $s_2 \xrightarrow{\beta} s_1$ and $s_3 \xrightarrow{\alpha} s_1$.

If both of these conditions hold for all states of an automaton A in which α and β are both enabled then α and β are said to be independent in A, otherwise they are defined to be *dependent*, both of which will be a key relationship when determining how to calculate ample sets.

2.5.2 Ample sets

The partial order reduction technique achieves reduction by identifying a subset of $enabled(s)$ for some state s which will be called $ample(s)$ [3,6]. Only the successor states of the events of $ample(s)$ are then used to construct the synchronous composition. If the number of events in $ample(s)$ is less than the number of events in $enabled(s)$ then what results is fewer states being added to the state space of the synchronous composition.

Following are a set of conditions known as the *ample conditions* that are taken from existing texts [3, 6], with one notable exception where we have replaced one of the conditions with a weaker version. These conditions are the criteria by which $ample(s)$ is selected. To show that the model produced by selecting a reduced number of states in this way preserves the properties of controllability and nonblocking of the full synchronous composition based on lock-step synchronisation as considered in this thesis, an original proof is offered in Section 3.2, proving that adhering to the ample conditions is sufficient when attempting to construct a reduced model that preserves properties of controllability and blocking.

For the following definitions let $A = \langle \Sigma, S, S^\circ, \rightarrow \rangle$ be an automaton and let $s \in S$

Definition 2.18. Condition **C1** is the *non-emptiness condition.*
If $enabled(s) \neq \emptyset$ then $ample(s) \neq \emptyset$

This first and simplest condition ensures that as long as there is at least one event enabled in state s, then there must be at least one event included in $ample(s)$.

Definition 2.19. Condition **C2** is the *dependency condition.*
For every transition sequence from s in automaton A, an event that is dependent on any event chosen for $ample(s)$ may not occur before an event from $ample(s)$.

Definition 2.20. Condition **C3** is the *cycle condition.*
A cycle consisting of states $S' = \{s_0, s_1, \ldots, s_n\}$ may only exist if
$\forall \alpha \in enabled(s_0) \cup enabled(s_1) \cup \ldots \cup enabled(s_n) : \alpha \in ample(s_i)$ for some $0 \leq i \leq n$

That is to say that a cycle may only exist in a reduced model if the combined set of all enabled events for the states on that cycle are at some point included in the ample sets of states on the same cycle.

Ensuring condition **C3** can prove to be challenging when constructing the algorithm, so a stronger condition **C3′** that ensures **C3** may be used instead. This stronger condition introduces the concept of a state being *fully expanded*.

Definition 2.21. A state s is defined to be *fully expanded* if $ample(s) = enabled(s)$.

Definition 2.22. Condition **C3′** is the *strong cycle condition*.
Let $S' = \{s_0, s_1, \ldots, s_n\}$ be a cycle, then there exists some state $s_i \in S'$ where s_i is fully expanded.

It has been shown in [17] that the conditions **C1**, **C2** and **C3′** are proven to preserve controllability. In this thesis we will replace **C3** and **C3′** with conditions **C4** and **C4′**, which are given in Chapter 3 which are subsequently proven in Section 3.2. If the ample conditions are satisfied when selecting events for each state then the resulting reduced model is guaranteed to preserve the properties of controllability and blocking of the full synchronous composition. This means that if M is uncontrollable or blocking then the reduced model M_R, generated by adhering to the ample conditions, will also be uncontrollable or blocking respectively. Likewise if M is controllable or non-blocking then so too will M_R be. A proof of this will also be given in Section 3.2.

Figure 2.11 gives an example of an automaton prior to reduction. We can try to reduce the state space in this example by applying the ample conditions when selecting $ample(s)$ for each successive state. First consider state s_0. It can be observed that $enabled(s_0) = \{\alpha, \beta, \sigma\}$. Determining the independence of these events gives us that α and β are independent and so are β and σ, while α and σ are dependent. Since β has the most independencies consider β first for $ample(s_0)$. **C1** is satisfied right away, **C2** is satisfied since β is independent of every other event, meaning no event dependent on β could be taken on any path before β itself is taken, and **C3** is satisfied since no cycles have been created yet. Since all ample conditions are satisfied, set $ample(s_0) = \{\beta\}$ and add transition $s_0 \xrightarrow{\beta} s_1$ to the reduced model. Now

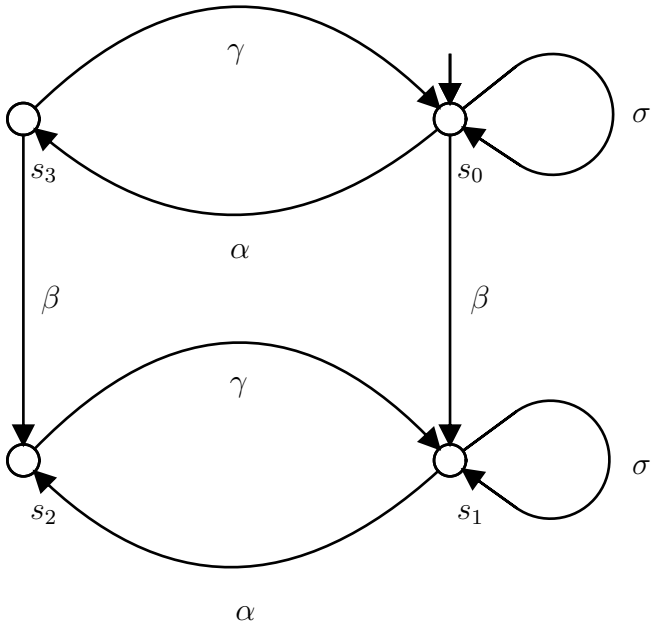

Figure 2.11: Automaton before reduction.

consider state s_1. It can be observed that $enabled(s_1) = \{\alpha, \sigma\}$ and we already know that those two events are dependent, so arbitrarily consider α first for $ample(s_1)$. **C2** is violated now since the path $s_1 \xrightarrow{\sigma} s_1$ exists from state s_1 which yields an event dependent on α occuring before α, so we cannot choose $ample(s_1) = \{\alpha\}$. Next consider $ample(s_1) = \{\sigma\}$. This violates **C3** as a cycle has been formed and $\alpha \in enabled(s_1)$ was not included in the ample set for any states on the cycle. This leads to the only remaining case where $ample(s_1) = \{\alpha, \sigma\} = enabled(s_1)$. Set $ample(s_1) = enabled(s_1)$ and add transitions $s_1 \xrightarrow{\sigma} s_1$ and $s_1 \xrightarrow{\alpha} s_2$ to the reduced model. Now consider state s_2. It can be seen that $enabled(s_2) = \{\gamma\}$, so condition **C1** gives us $enabled(s_2) = \{\gamma\} = enabled(s_2)$ immediately. Set $ample(s_2) = enabled(s_2)$ and add transition $s_2 \xrightarrow{\gamma} s_1$ to the reduced model. Since no more states were added with this transition the reduced model is complete. Figure 2.12 shows the reduced model.

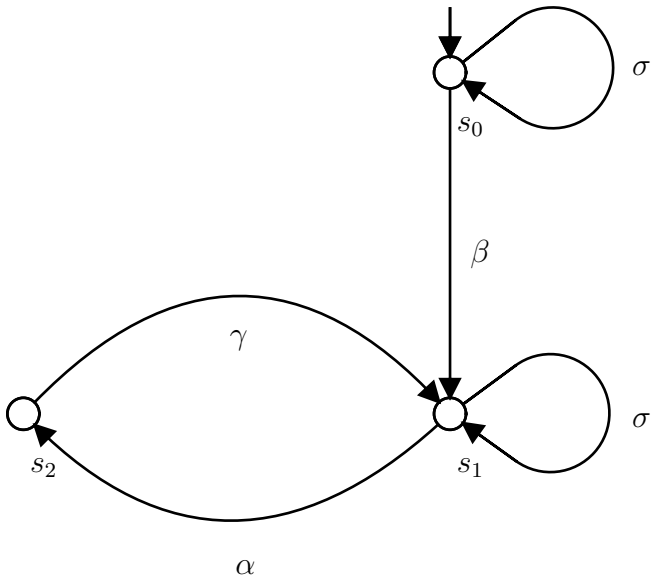

Figure 2.12: Automaton after reduction subject to ample conditions.

2.6 Compositional Verification

Compositional verificaton is another technique that is used to try and address the state space explosion problem. As the problem generally arises when dealing with the composition of several automata, a sensible approach would seem to be reducing the state space in the individual components before composition. Once we have established that this can be done, we can compose automata two at a time, and once again perform the simplification step on the result. These resulting simplified automata can then be subjected to the same process and be composed two at a time and again simplified. This can be done until the system has been simplified to a point that verification can be carried out in the conventional way.

Obviously key to this process are the methods by which the automata can be simplified. This needs to be done safely so that the resulting abstractions will still remain conflict equivalent to the original automata. As such there are several rules which can typically be applied, each of which attempt use a

different approach to reduce the state space by identifying different properties of the states. There is however no universally determined method to obtain the greatest reduction for an arbitrary automaton. Since this is the case it is not required for compositional verification to use any particular combination of reduction rules, rather that as much reduction as possible is achieved using whichever rules are appopriate while limiting the computational overhead of doing so.

These reduction rules operate fundamentally by exploiting the behaviour of the system when *silent transitions* occur.

Definition 2.23. The *silent event* τ is a special event used in each of the rules. Typically τ is not included in the event alphabet for an automaton, however when it is useful to do so the event set is referred to by Σ_τ. A transition involving a silent event is known as a *silent transition*. Such transitions are identified as they can be performed in the automaton in which they appear without having any effect on the state of any other automaton.

Definition 2.24. In order to make use of this silent event τ, abstractions of automata are created where certain events are removed from the alphabet and all transitions involving those events replaced by τ transitions. This process is knows as *hiding*.

In order to achieve this effect of creating silent transitions, the events that are hidden by the hiding process are often the *local events* of an automaton, those being the events of an automaton that do not appear in any other automaton.

Definition 2.25. When considering τ as part of a string it is often useful to be able to refer to that string with the τ events removed. If $p = \tau^*\sigma_1\tau^*\sigma_2\tau^*\ldots\tau^*\sigma_n\tau^* \in \Sigma_\tau^*$, then $P_\tau(p)$ denotes the string $q = \sigma_1\sigma_2\ldots\sigma_n$, that is, the string p with the τ events removed. Conversely $s \stackrel{q}{\Rightarrow} t$ implies the existence of string p such that $P_\tau(p) = q$ and $s \stackrel{p}{\rightarrow} t$, which is to say that $\stackrel{p}{\rightarrow}$ denotes a path with exactly the events of q, whereas $\stackrel{p}{\Rightarrow}$ denotes a path with an arbitrary number of τ events inserted into the string p.

In order to safely perform the reduction rules there must be some qualitative way to determine whether or not the abstracted automaton achieved post reduction is still equivalent to the original automaton. The compositional verification process can be used to reduce models for the purposes of controllability [20] or nonblocking verification. For the purposes of this thesis during compositional verification we concern ourselves with the property of nonblocking. We then introduce the concept of *conflict equivalence* [**?**]. Conflict equivalence is an equivalence relationship that determines whether two automata, which when composed with an arbitrary automaton, will yield the same result of blocking or nonblocking. The arbitrary automata used for the compositions are called *tests* and will often be referred to as an automaton T when conflict equivalence is being established.

Definition 2.26. Two automata G_1 and G_2 are considered conflict equivalent, $G_1 \simeq_{conf} G_2$ if, for any test T, $G_1||T$ is nonblocking if and only if $G_2||T$ is nonblocking.

The application of the rules are at a state level in a single component automaton. This is to say that the states in a single component automata are examined in an attempt to identify states that exhibit certain properties with respect to the silent event. These states can then have the appropriate rules applied to them repeatedly. The criteria that the states must satisfy for the purposes of this thesis pertain to conflict equivalence. This means that any states selected for the rule application must not have any future behaviours that can be distinguished by conflict equivalence. Such states are known as *conflict equivalent states* and may be merged without affecting possible conflicts with other components.

While there are several rules that may be applied during compositional verification, this thesis attempts to introduce a variation of just one of those rules, the *silent continuation* rule [8]. To effectively describe silent continuation a some more notation must be introduced.

Definition 2.27. A *stable state* is a state without any outgoing τ transitions.

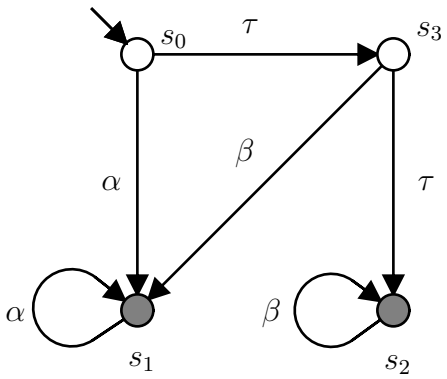

Figure 2.13: Automaton before application of the silent continuation rule.

Definition 2.28. Let $G = \langle \Sigma, S, S^\circ, \rightarrow, Q \rangle$ be an automaton. The relation $\simeq_{inc} \subseteq S \times S$ is defined such that $s \simeq_{inc} s'$ if

$$S^\circ \stackrel{\varepsilon}{\Rightarrow} s \iff S^\circ \stackrel{\varepsilon}{\Rightarrow} s'; \tag{2.1}$$

$$\forall t \in S, \forall \sigma \in \Sigma : t \stackrel{\sigma}{\Rightarrow} s \iff t \stackrel{\sigma}{\Rightarrow} s' \tag{2.2}$$

If this holds s and s' are considered *incoming equivalent*.

The silent continuation rule states that if two states are incoming equivalent, and they can each reach one or more stable states using a nonempty sequence consisting entirely of silent transitions, then those two states are conflict equivalent. The idea here is that in essence, the silent transitions do not have any effect on any tests that could be introduced. As such the only transitions that matter are the non τ transitions. Ensuring that the states can always be reached by the same sequence of non τ events ensures that the same sequence will be able to reach the merged state. Similarly, since the silent transitions have no effect on the tests, only the outgoing events from the stable states are required in order to reach marked states. As such the τ transitions may be collected together into the merged state as a single τ transition to the stable states.

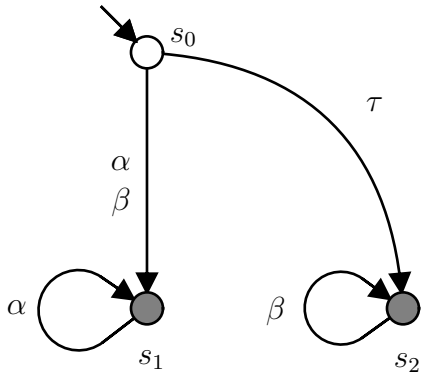

Figure 2.14: Automaton after application of the silent continuation rule.

Figures 2.13 and 2.14 give examples of an automaton before and after the application of the silent continuation rule respectively. On application of this rule states s_0 and s_3 have been merged. It can be seen in Figure 2.13 that states s_0 and s_3 can both be considered initial, since both can be reached with 0 or more silent transitions, satisfying the first condition for the application of the silent continuation rule. Also they can both reach state s_2 using a nonempty sequence of τ transitions. State s_2 is observed as being a stable state, as its only outgoing transition is using event β. This satisfies the second condition for the application of the silent continuation rule. Since all of the silent continuation rule conditions are met by states s_0 and s_3 they can be merged into a single state, shown as state s_0 in Figure 2.14.

This rule will serve as the basis for the developments in Chapter 4. Some of the requirements are altered slightly so a proof is offered in the same chapter.

Chapter 3

Partial Order Reduction With Tarjan's Algorithm

This chapter will introduce and discuss the work done in adapting the existing work of partial order reduction in discrete event systems for safety properties. Several areas of improvement have been explored and this has prompted the development of new algorithms and proofs.

Section 3.1 will detail the various algorithms that were created or improved upon to realise the research done in this area. This will include both a description of the algorithms along with the motivations for them and the areas in which they improve upon existing work wherever it exists. Section 3.2 will provide proofs for the various conditions, concepts, and algorithms developed in this chapter. The work done in [17] introduced an algorithm for partial order reduction in discrete event systems for controllability. We build on this research by extending the algorithm to also work for nonblocking. As such some of the proofs offered in this thesis will be altered versions of those offered in [17], while some will be original proofs, the distinction of these will be made clear in Section 3.2. Section 3.3 will provide the experimental results of the research in this area along with analysis. A discussion of expected results versus achieved results will also be included. Section 3.4 will summarise the research done, reflect on what was achieved and discuss the possibilities for further research in this area.

Before beginning a discussion on the algorithms developed for this chapter it will be valuable to first discuss which various areas of improvement were discovered and to introduce any new concepts that resulted.

It was noticed that it was possible for a potentially fewer number of states to be fully expanded in order to comply with cycle condition **C3′**. The motivation behind this was the realisation that the logic used in the original proof of correctness for condition **C3′** satisfying **C3** could extend to strongly connected components instead of being limited to just cycles. This leads to the introduction of two new ample conditions particular to this section.

Definition 3.1. Condition **C4** is the *component condition*.
A component consisting of states $S' = \{s_0, s_1, \ldots, s_n\}$ may only exist if $\forall \alpha \in enabled(s_0) \cup enabled(s_1) \cup \ldots \cup enabled(s_n) : \alpha \in ample(s_i)$ for some $0 \leq i \leq n$

Definition 3.2. Condition **C4′** is the *strong component condition*.
Let $S' = \{s_0, s_1, \ldots, s_n\}$ be a component, then there exists some state $s_i \in S'$ where s_i is fully expanded.

Conditions **C4** and **C4′** have been introduced to replace the cycle conditions **C3** and **C3′** respectively, which are offered by [3,6].The component conditions are weaker versions of the cycle conditions, which we predict should be able to allow a greater number of reduced ample sets at the cost of the computational overhead of calculating strongly connected components. This reasoning for this is that each cycle belongs to a strongly connected component. If every cycle were to contain a fully expanded state, which is the requirement of condition **C3′**, this could result in several fully expanded states on one component, in order to satisfy condition **C3′**. If we instead require only one fully expanded state per strongly connected component, this then allows for potentially less states to be fully expanded. This would be a more optimal result as it would result in fewer transitions being added to the reduced model, which in turn has the potential to result in fewer states being created.

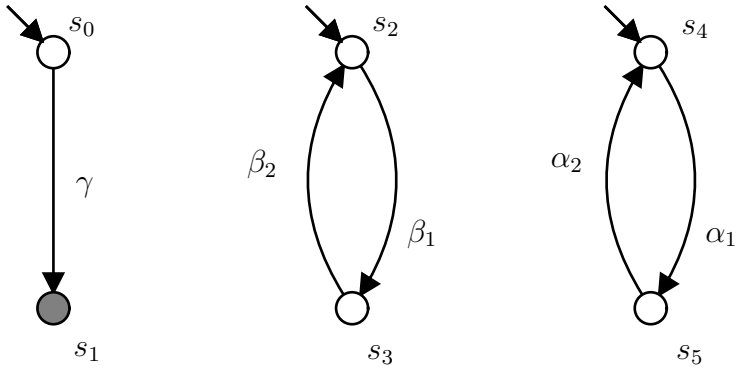

Figure 3.1: An example system to demonstrate the strict component condition.

An example is provided in Figures 3.1, 3.2 and 3.4. The system in Figure 3.1 depicts the system that will be reduced. The first automaton has just two states with a γ transition between them. Notice that state s_1 is the only marked state in the system, so unless a γ transition occurs, the system cannot reach a marked state, and furthermore once a γ transition has occurred the system cannot leave a marked state. The other two automata are simple two state cycles, using α and β transitions. Notice that all of the events in this system are independent, thus any choice of event for the ample sets when constructing the reduced model is guaranteed to satisfy the dependency condition **C2**. Figure 3.4 shows the complete synchronous product for this system. It can be seen that this synchronous composition has 18 distinct cycles and two strongly connected components. Let component $C_1 = \{s_0, s_2, s_4, s_6\}$ and $C_2 = \{s_1, s_3, s_5, s_7\}$. Notice that the only way to get from C_1 to C_2 is by taking a γ transition. As this is the case, the resulting markedness of the components has all the states in C_2 marked whereas the states of C_1 are all unmarked. To illustrate the necessity of the component and cycle conditions **C4** and **C3** respectively, consider constructing the reduced synchronous product without using either condition.

From the initial state s_0 there are three enabled events $enabled(s_0) =$

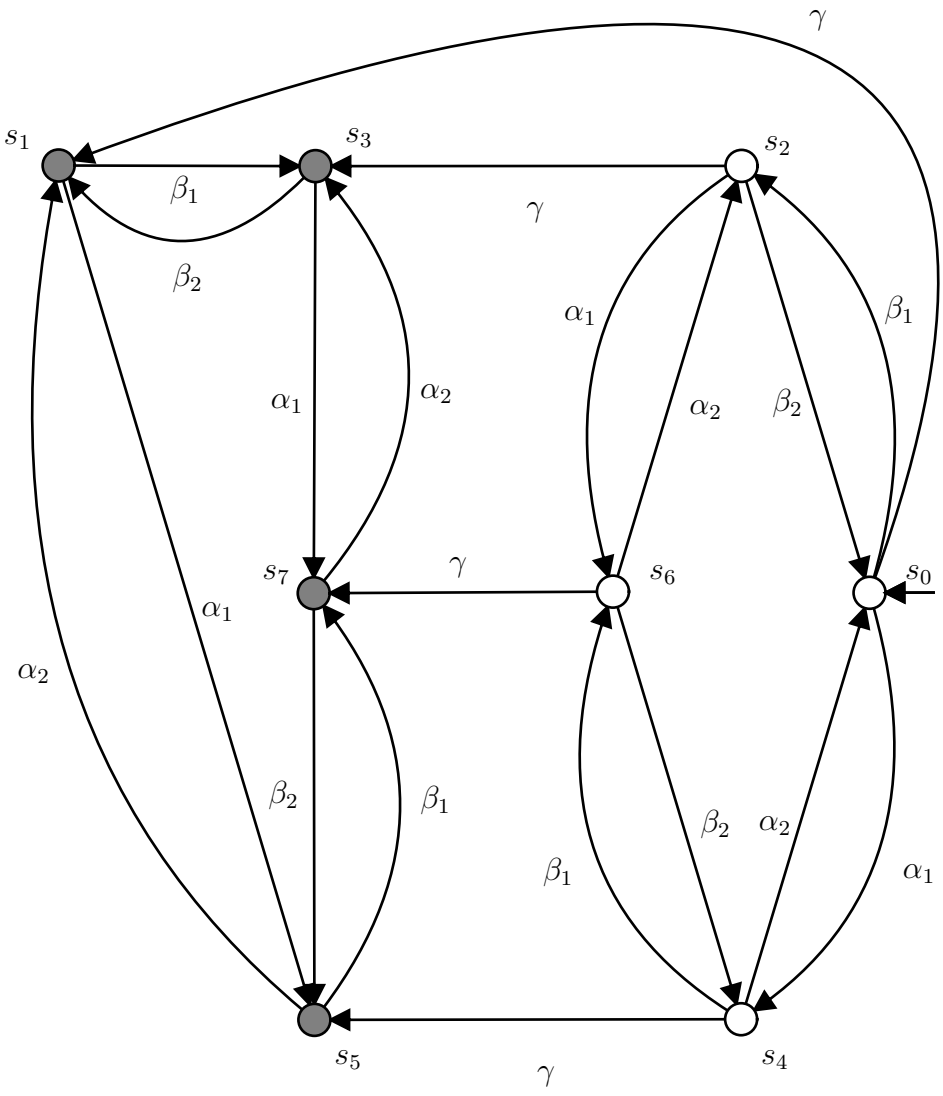

Figure 3.2: Synchronous composition of the automata given in Figure 3.1.

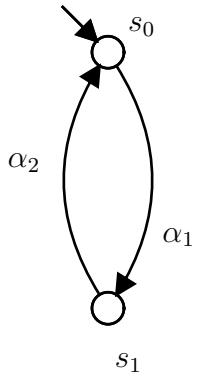

Figure 3.3: Synchronous composition without **C3** or **C4**.

$\{\gamma, \alpha_1, \beta_1\}$. Since all events are independent, it is safe to choose any event for $ample(s_0)$. Let us choose $ample(s_0) = \{\alpha_1\}$. Following that transition we reach state s_4. Similarly as for state s_0, we can choose $ample(s_4) = \{\alpha_2\}$, which takes us back to s_0, completing the algorithm. Figure 3.3 shows the resulting reduced automaton when the component condition is not used. Clearly this does not retain the property on nonblocking of the full synchronous composition as there are no reachable marked states. The problem arises as since γ and β_1 are independent of the choices for ample, the γ and β_1 transitions are always deferred off to a later stage. This is a problem because, as we noticed, a cycle was closed before the γ and β_1 transitions were added, which effectively meant they were ignored.

To remedy this we can impose the strict cycle condition **C3′**. What will now be shown is that without careful selection of which states should be fully expanded, many more states than necessary can end up being added to the reduced synchronous composition. When using the cycle condition, the final step in the previous example is noticed to have closed the cycle $\{s_0, s_4\}$, so one of the states on the cycle must be fully expanded. Suppose we choose to fully expand the state s_0, this then adds states s_2 and s_1 to the reduced automaton. Let us consider for the moment constructing the ample set for s_2. We have $enabled(s_6) = \{\gamma, \alpha_1, \beta_2\}$, for the same reasoning as above we

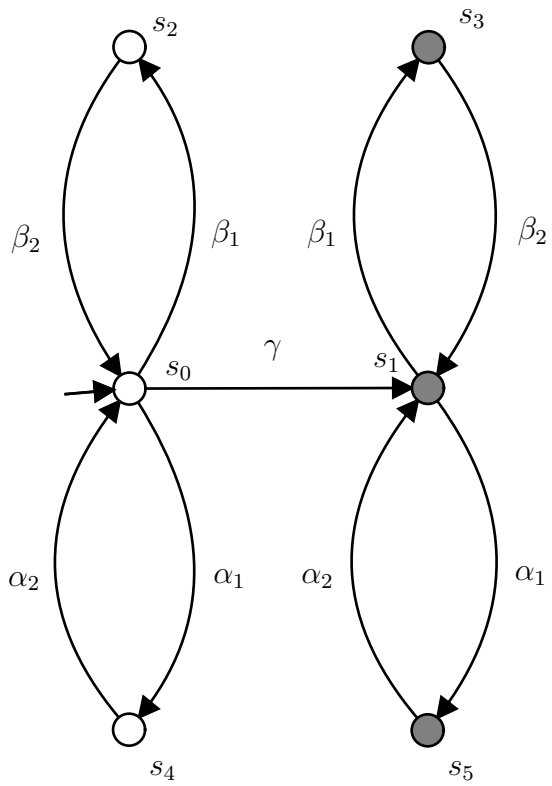

Figure 3.4: Automaton constructed using the strict component condition instead of the strict cycle condition.

can choose $ample(s_6) = \{\beta_2\}$. Once again we notice that a cycle has been created, namely $\{s_2, s_0\}$, and we must fully expand a state on this cycle. The problem that arises here is that every state in this graph is part of a cycle, so every time a state is fully expanded and new states are included, they too in turn may find themselves to be part of a cycle and fully expand, until the reduced synchronous product is no different than the full synchronous product shown in Figure 3.2.

Now let us construct the reduced automaton using the strict component condition **C4′** instead of the strict cycle condition **C3′**. Using the previous example to the point where the cycle $\{s_2, s_0\}$ was created, we now can use the

fact that cycles $\{s_2, s_0\}$ and $\{s_4, s_0\}$ belong to the same strongly connected component C_1. Since this is the case, and the component rule demands that just one state per strongly connected component must be fully expanded, we are under no obligation to fully expand any more states. Since we have now finished exploring state s_0, all that remains is to explore state s_1 that was reached on the addition of γ to $ample(s_0)$. We can notice here that if we ignore the γ transitions, states s_0 and s_1 share the same structure, each having a $\alpha_1 + \alpha_2$ cycle with states s_4 and s_5 respectively; and a $\beta_1 + \beta_2$ cycle with states s_2 and s_3 respectively. As such, continuing the creation on the reduced automaton ends up creating both of these cycles from s_1. Again however once the second cycle is created we notice that a state in component C_2 has already been fully expanded, and thus the algorithm can terminate without adding any new states. The resulting reduced automaton is given in Figure 3.4.

What we can observe here is that while a careful selection of which states to fully expand could achieve the same result with the cycle condition as with the component condition, the component condition guarantees the result. As models typically contain a very large number of cycles and usually only one or two strongly connected components, the computational overhead involved in implementing the two methods is far larger in the case of the cycle condition. Where the component condition need only track how many components have been created and whether or not they contain a fully expanded state, which may often simply occur accidentally, the cycle condition must keep track of all fully expanded states throughout the entire process so that it may be checked whether or not a state in a current cycle has already been fully expanded. In very large systems this can often be a significant memory allocation, which would be counter to the goal of the overall process.

Another area in which the component condition yields improvements over the cycle condition lie in the algorithm used to detect each of them. The algorithm originally detailed was simply an approximation for cycle detection. That is that while every state that was part of a cycle was definitely

detected as such, it was also the case that there could potentially be some states that were determined as being part of a cycle, despite that not being the case. Because of this there were improvements to be found by more accurately detecting these cases which is handled by the component condition. It will be shown in Section 3.1 that the algorithm developed to handle the component condition determines exactly all of the components in the reduced automaton, so no over-approximation of states is ever required.

All of this does of course hinge on the fact that these new conditions are in fact still correct, as the cycle conditions have been proven to be. As mentioned on page 30 the correctness of these conditions will be proven in Section 3.2.

It was also noticed that when selecting the events for the ample sets, using some criteria might help yield smaller ample sets in general, which again could result in greater reduction in the state space of the reduced model. The existing strategy for selecting events just used a greedy approach of events ordered the same way as they were originally passed in to the algorithm. It seemed quite likely that this was not optimal so several different selection strategies were attempted instead. One of these strategies was to order events according to the number of independencies that they shared with other events. By ordering events this way we predicted that it should be easier for the ample sets to satisfy the dependency condition **C2**, leading to generally smaller ample sets overall. The reasoning behind this was that by selecting an event with the maximum possible number of independencies, we are simultaneously selecting an event with the minimum number of dependencies. This should make it less likely to encounter an event that depends on one of the events chosen for ample on a path from the current state, before encountering one of those ample events on the same path, which is what condition **C2** states.

Another selection strategy was to order events dynamically on a state by state basis depending on whether or not the events will take us to a state

that has already been visited. This ordering would need to be determined as a state was being explored due to the fact that the target state for an events depends upon the source state. The strategy would put events that take us to states that have previously been visited before events that would make us visit new states. The reasoning behind this was that if at all times we are attempting to make sure that we do not visit new states unless we have to, this should lead to a smaller state space in general. There is the possibility however that forcing this ordering could lead to suboptimal ample sets, which would then require adding additional events which could lead to the creation of new states. This suggests that some combination of the independence selection strategy and this one could be a better approach.

3.1 Algorithms

This section introduces the algorithms developed and improved upon in order to implement the partial order reduction model verifier described in this thesis. In particular a verifier for nonblocking is a new development in this research while the controllability verifier builds upon the work done in [17]. As this suggests, there will be two versions of this verifier implemented, a controllability verifier and a nonblocking verifier, that vary with regard to the property that they are verifying. Since these model verifiers will be used to verify the properties of controllability and nonblocking in the models that they attempt to solve, the algorithms have been designed in such a way that those properties of the original models are preserved in the models that are generated as a result of the partial order reduction process. Once a verifier has completed the verification process, a result of either true or false is returned signifying that the given model was nonblocking or not in the case of the nonblocking verifier, or if it was controllable or not in the case of the controllability verifier.

The work in [17] offered a depth first search implementation of the partial order reduction process to determine controllability. We have now generalised and improved upon this so that the partial order reduction process can also

answer questions pertaining to nonblocking, as well as utilising strongly connected components to satisfy the ample conditions. This means that now these algorithms no longer attempt to satisfy ample condition **C3** or **C3′**, but instead will satisfy the weaker conditions **C4** and **C4′**. Due to the difference in the properties of controllability and nonblocking, the verification of these properties must be handled quite differently. Since controllability is determined by whether or not a particular state enables certain uncontrollable events, it can be determined on a state by state basis if a model is controllable. This means that, as in the existing work, controllability is determined as each state is expanded just before selecting the ample set. Nonblocking however is a question of reachability and as such can not be determined on a state by state basis. Instead we must determine if all reachable states can reach a marked state, a question which typically requires the full state graph to answer. We can however introduce a new criterion for determining nonblocking using strongly connected components, which allows us to verify nonblocking during the depth first search.

Definition 3.3. Let $G = \langle \Sigma, S, S^\circ, \rightarrow, Q \rangle$ be an automaton. G is nonblocking if and only if for all strongly connected components $C \subseteq S \in G$, $\exists s \in C, t \notin C : s \in Q \vee s \rightarrow t$

This means that if a component is created that does not contain a marked state, then then it must be able to reach another component, or the model is blocking. We introduce an iterative implementation of Tarjan's algorithm [19] in order to calculate the strongly connected components. Tarjan's algorithm offers a linear time solution to finding the strongly connected components in a graph. More specifically the complexity of the algorithm is $O(s+t)$ where s is the number of states in the graph and t is the number of transitions in the graph. Tarjan's algorithm operates by making two interleaved traversals of the state space. First, a depth first traversal of the state space is performed. As this is happening, states are added to a stack and they are marked with a root index which represents to index of the component which they belong to. Initially this root index is equivalent to the depth first index of the state, the order in which the state was visited during the depth

first traversal, it is then updated to be the minimum index of any successor state that has already been visited and has not already been determined to be in a component. Once a component has been found, all states that belong to that component are marked as belonging to that component. This is accomplished by removing states from the stack until encountering a state whose component root index is equivalent to its depth first index. Those states then form the component indexed by that component root index.

3.1.1 Tarjan's Algorithm

In this section we will describe in more detail how Tarjan's algorithm operates in order to determine the strongly connected components of a system. Note that since Tarjan's algorithm applies generally to all graph theory and not just model checking, the terminology used differs somewhat to that which has been introduced in this thesis so far. As such for the remainder of this section, a *node* can be considered to be a state, and an *edge* can be considered to be a transition. Algorithm 1 gives the psuedocode for Tarjan's algorithm.

The algorithm is comprised of two functions, $MAIN$ which is used to make sure every state is visited, and $VISIT$ which is recursively called in order to perform the depth first traversal and calculate the strongly connected components. This algorithm uses to idea of a *root* node for each component. Each component is designated one node that is to serve as its *root* node. The root can be considered as the "starting point" for a component. There are a number of global variables used in order to keep track of various things during the traversal. Global variable *stack* stores the states that are being explored in the depth first traversal and is the main mechanism behind the depth first search. Global variable $root[]$ is an array of nodes, indexed by nodes. This is used in order to determine which node is the root of the strongly connected component that the node being used to index the array belongs to ie. $node[v] = w$ indicates that the root of the strongly connected component to which node v belongs, is node w. Finally

Algorithm 1 Tarjan's algorithm

```
1: function VISIT(v)
2:     root[v] = v
3:     InComponent[v] = false
4:     PUSH(v, stack)
5:     for all nodes w where v → w do
6:         if w is not already visited then
7:             VISIT(w)
8:         end if
9:         if ¬InComponent[w] then
10:            root[v] = MIN(root[v], root[w])
11:        end if
12:    end for
13:    if root[v] = v then
14:        repeat
15:            w = POP(stack)
16:            InComponent[w] = true
17:        until w = v
18:    end if
19: end function
20: function MAIN
21:    stack = ∅
22:    for all nodes v ∈ V do
23:        if v is not already visited then
24:            VISIT(v)
25:        end if
26:    end for
27: end function
```

global variable $InComponent[]$ is a boolean array indexed by nodes used to determine whether or not a node has already been determined to be part of a component ie. $InComponent[v] = true$ states that node v has been determined to be part of a component already. The goal of the algorithm is to determine the root of every component, and hence calculate all the components of the system.

The algorithm begins in $MAIN$ by first initialising *stack* to store the states in the depth first traversal. There is then a call to the recursive method $VISIT$ for every state that has not already been visited. This ensures that even the strongly connected components comprised of unreachable states are calculated, as otherwise the depth first search made by $VISIT$ would only visit states that are reachable from the initial state. The rest of the algorithm takes place in the $VISIT$ method.

The $VISIT(v)$ method operates as follows. Initially we set $root[v] = v$ and $InComponent[v] = false$, meaning that the node being visited is initially considered to be the root of the component to which it belongs, and also not yet part of any found component. Node v is then pushed onto the stack. Next, every successor node w from the edges of node v is visited with the recursive call $VISIT(w)$. This begins the depth first traversal, as this process is then repeated on node w, and each subsequent node from there. Once the $VISIT$ routine completes on node w, we are ensured that all reachable nodes from node w have been visited. Some of these nodes may have been visited prior to node w in depth first order, as such this would indicate that node w, and hence node v, belongs to a component whose root is at least one of those earlier visited nodes. The next step sets the root of node v to be the result of $MIN(root[v], root[w])$, where $MIN(x, y)$ returns x where x appears before y in depth first order, otherwise it returns y. This effectively finds the minimum node in terms of depth first order that is reachable from node w, if that state appears earlier than node v, then node v must belong to a component whose root is the minimal node that was found, since node v can reach node w. To see how this works, consider the example given in Figure

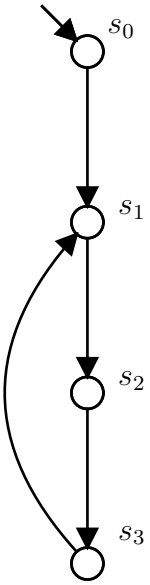

Figure 3.5: Example automaton to demonstrate Tarjan's algorithm.

3.5.

First observe that this automaton has two strongly connected components $C_1 = \{s_0\}$ and $C_2 = \{s_1, s_2, s_3\}$. Initially we call $VISIT(s_0)$ which sets $root[s_0] = s_0$. Then we recursively call $VISIT$ on nodes $s_1 \ldots s_3$ each of which set $root[s_i] = s_i$. In $VISIT(s_3)$ we notice that the successor of s_3 is s_1. Since s_1 is already visited we do not make another recursive call, and now we have run out of successors so the recursion can start to return back. Before that happens however, we have to set $root[s_3] = s_1$, as s_1 was a successor and $MIN(s_3, s_1) = s_1$ since s_1 appears before s_3 in depth first order. We now return back to the $VISIT(s_2)$ call, and set $root[s_2] = root[s_3] = s_1$. Returning back to $VISIT(s_1)$ now we notice that $root[s_2] = s_1$, which is the node we are visiting. This indicates that all of the nodes reached from the edge that visited node s_2 belong to the same component as node s_1. Since node s_1 has no other outgoing edges, this means that we have found the component C_2. Returning back to the $VISIT(s_0)$ call we have $MIN(root[s_0], root[s_1]) = root[s_0] = s_0$, which again is the node we are

visiting. Again since there are no more outgoing edges from node s_0, we have found the last component C_1.

We see from this example that once we have visited all of the successors of a node and the root for that node is still equivalent to that node, then we have found a strongly connected component. All that remains is to create the strongly connected component, which is done by setting $InComponent[v] = true$ for every node v in the component. This is done after the loop visiting each of the successors. First we check if $root[v] = v$. As previously stated, if this condition is true then we have found a strongly connected component with its root of node v, so we can begin to create the component. We then pop nodes from the stack and set them in component, until we pop the root itself, at which point the strongly connected component has been created. As the nodes are being popped we are guaranteed that they all belong to the same strongly connected component. This is because the components are found in depth first order, so any nodes belonging to other components will have already been popped, or have not yet been visited by the algorithm and hence have not yet been added to the stack.

3.1.2 Depth First Search with an Iterative Tarjan's Algorithm

In this section we will introduce an iterative implementation of Tarjan's algorithm. This iterative implementation of takes the principles of Tarjan's algorithm, but removes the need for recursive calls. This was necessary as the heavily recursive nature of Tarjan's algorithm would have severe memory requirements in keeping track of the function frame stack when attempting to calculate the strongly connected components in large models. By translating the algorithm to an iterative approach we eliminate these resource requirements which then allows us to use strongly connected components during the partial order reduction process. Algorithm 2 gives the psuedocode for the iterative implementation of Tarjan's algorithm.

We again use the idea of a component root in this algorithm, but this time we need another stack in order to keep track of which states in the depth first search has been fully explored and hence can be added to a strongly connected component. We still have the original component stack, so now we have two stacks called the *requestStack* and the *componentStack*. The items added to the *requestStack* are tuples of the form $(requestType, prev, state)$, where $requestType$ may be either $CLOSE$ or $VISIT$, and *prev* and *state* are both states. These tuples represent a transition in the system and whether or not the successor state of that transition has been fully explored. If the request type is $CLOSE$ then this indicates that the *state* variable for this transition has been fully explored, whereas $VISIT$ indicates that this tuple is still part of the current forward search. The *state* variable represents the successor state of the transition whereas *prev* represents to source state of the transition. We keep all of this information in the tuples so that the roots of states found in the depth first traversal may be passed back down the same path that was taken. This *requestStack* mimics the recursion that is present in Tarjan's original algorithm.

The algorithm begins by pushing the tuple $(VISIT, null, init)$ to the *requestStack*, where *init* is the initial state of the model. We then begin the loop that performs depth first traversal which will add tuples to the *requestStack*, so that once the *requestStack* is empty we should have computed all of the strongly connected components. During the loop we always have access to the variables *prev* and *state* which have been detailed above. We also have the *req* variable which stores the $requestType$ of the current tuple being considered. The first part of this loop is the conditional on line 5 which checks whether or not $req = VISIT \wedge state$ not visited. If this is true, then the current tuple represents a forward transition in the depth first search to a state that has not yet been visited, as such the state must be explored. To explore the state, first we set $state.visited(true)$. This ensures that any subsequent transitions to *state* will not result n *state* being explored again, as the above condition would be violated. We also set $root[state] = state$ meaning that initially every state is considered to be the root of the strongly

Algorithm 2 Iterative Tarjan's Algorithm

```
1: function FindComponents(init)
2:     requestStack.push(VISIT, null, init)
3:     while requestStack ≠ ∅ do
4:         (req, prev, state) = requestStack.pop()
5:         if req = VISIT and state not visited then
6:             state.setVisited(true)
7:             root[state] = state
8:             if state has successors then
9:                 inComponent[state] = false
10:                componentStack.push(state)
11:                requestStack.push(CLOSE, prev, state)
12:                for all states succ where state → succ do
13:                    requestStack.push(VISIT, state, succ)
14:                end for
15:            else
16:                inComponent[state] = true
17:            end if
18:        else
19:            if prev ≠ null then
20:                if ¬inComponent[state] then
21:                    root[prev] = min(root[prev], root[state])
22:                end if
23:                if requestStack.top() ≠ (VISIT, prev, ?) then
24:                    if root[prev] == prev then
25:                        repeat
26:                            w = componentStack.pop()
27:                            inComponent[w] = true
28:                        until w == prev
29:                    end if
30:                end if
31:            end if
32:        end if
33:    end while
34: end function
```

connected component to which it belongs, as was the case with the original algorithm. We then have $componentStack.push(state)$ which means that this state will, at some point when the strongly connected component to which $state$ belongs, be able to be added to that component. Next we push the tuple $(CLOSE, prev, state)$ followed by $(VISIT, state, succ)$ for each successor $succ$ of $state$. This ensures that by the time we pop the $(CLOSE, prev, state)$ tuple, we must have already popped each of the successor tuples and performed the exploration on them, meaning that at that moment we are ready to consider the strongly connected components to which $prev$ and $state$ belong. The manner in which this consideration is made will be explained when we examine from line 21. Note that all of this is only done if there exist successors for $state$. If not then we simply set $inComponent[state]$ and add nothing to either stack. This is because by definition a state with no successors must belong to a one state component, consisting of that state. As such $state$ need not be added to the $componentStack$, nor must a $(CLOSE, prev, state)$ tuple be added to the $requestStack$, as the component can just be created immediately, so there is effectively no component left to close. This brings us to line 21. At this point we either have that $req == CLOSE$ or that $state$ has already been visited. In either case we are ready to determine the minimum root of $prev$. This is either going to be some state that occurs before $prev$ in the depth first ordering, or $prev$ itself. In the former case, let us refer to this minimum root as $minRoot$. In the $req == CLOSE$ case we have finished exploring from $state$, and so if it has been found that a state $minRoot$ with a smaller root than $prev$ could be reached, this information will have been passed down to $state$, and since $prev$ can reach $state$, the minimum root for $prev$ must be $minRoot$. In the case that $state$ has already been visited, we have arrived at a point where it may be the case that $root[prev]$ represents a state with a greater depth first index than $root[state]$. As such if this is the case then we can safely update the root of $prev$ to be the root of $state$. Regardless of which case we fall in to however, if it is the case that $prev$ is $null$ then we have must have popped off the tuple $(CLOSE, null, init)$, meaning that the initial state has been fully explored and hence all components found, so we can skip to the end of the main loop. That case aside,

we then check if $\neg inComponent(state)$. If this is not true then the current tuple represents a transition to another strongly connected component containing *state*. This means that *prev*, which is part of the current forward search, must be a part of a different strongly connected component, since every state in *state*'s component has already been determined. As such we do not need to reconsider the root of *prev*, as only states that can be a part of the current strongly connected component are valid choices for $root[prev]$. If however it is not the case that $inComponent(state)$ then it must be the case that *state* and *prev* belong to the same strongly connected component. Since this is the case we then set $root[prev] = min(root[prev], root[state])$ for the reasons detailed above. Once we have determined the correct root of *prev* we then check if $requestStack.top() \neq (VISIT, prev, any)$ on line 26. By this we are determining whether or not there are still states to explore from *prev* in the current forward search. The *any* state represents an arbitrary state. A false result would mean that there may still be states to be considered for this strongly connected component, so the traversal must first explore those states. A true result however means that from *prev*, there are no more states to consider for this strongly connected component. As such we can then check to see if all of the states for this component have been found. We do this by checking if $root[prev] = prev$. If this is the case, and the root of prev has not been changed, then no state with a smaller depth first index than *prev* was reachable from *prev*, as such *prev* must be the root of this strongly connected component. We then create the component by popping states from the *componentStack* and marking them *inComponent*, until *prev* has been popped from the *componentStack*.

It can be seen from this algorithm that when states are re-visited and smaller roots are found in the forward search, these smaller roots are then passed back through all of the previous states on the depth first path, until the root itself is being considered in a $CLOSE$ tuple. Note that not all states in the strongly connected component necessarily have a reference to the actual component root, it is only necessary that the states determine that they are not themselves the root and pass that information back.

Using this algorithm as a base, we can now perform the partial order reduction process with the strict component condition **C4′**. We can then check the resulting models for either nonblocking or controllability. To perform the partial order reduction process we must calculate the ample set of events for the states when we are exploring the successors. Also before we create a strongly connected component we must ensure that it contains at least one fully expanded state. Algorithm 3 gives the psuedocode for determining controllability in models using partial order reduction and Tarjan's algorithm, and Algorithm 4 shows the same but for nonblocking.

Since it is effectively the same in both cases, we will use Algorithm 3 to explain how the partial order reduction is performed and the same description will apply for Algorithm 4. On line 12 we notice that we make a call to $ample(state)$ in order to return the ample set of events, stored in the collection *ample*, for *state*. In the special case for the controllability checker, we also check to see if this set returned is null, in which case we have determined that *state* in in fact uncontrollable, so we return false. If this is not the case however, we then use the events from *ample* to push on the new tuples. In addition to this on line 28 we check to see if the strongly connected component that has been detected contains a fully expanded state with a call to $fullyExpanded()$. As we are calculating the ample sets we also check to see if for any state s, $ample(s) = enabled(s)$, at which point we flag $s.fullyExpanded(true)$. Because of this we can easily check if there is a fully expanded state in the strongly connected component. If the component does not contain a fully expanded state, then we are not ready to create the component yet. We then take the root of the component *prev* and expand it using $enabled(prev)$ instead of $ample(prev)$, ensuring that *prev* is fully expanded and that the component contains a fully expanded state. Again just for the controllability verifier, we return false if this call to $expand(prev)$ returns null. Any new states will then be added to the *componentStack* and included in the component creation the next time the component is closed.

Algorithm 3 Iterative Tarjan's Algorithm with Partial Order Reduction for Controllability

```
1: function IsControllable(init)
2:     requestStack.push(VISIT, null, init)
3:     while requestStack ≠ ∅ do
4:         (req, prev, state) = requestStack.pop()
5:         if req = VISIT and state not visited then
6:             state.setVisited(true)
7:             root[state] = state
8:             if state has successors then
9:                 inComponent[state] = false
10:                componentStack.push(state)
11:                requestStack.push(CLOSE, prev, state)
12:                if (ample = ample(state)) = null then
13:                    return false
14:                end if
15:                for all states succ ∈ ample do
16:                    requestStack.push(VISIT, state, succ)
17:                end for
18:            else
19:                inComponent[state] = true
20:            end if
21:        else
22:            if prev ≠ null then
23:                if ¬inComponent[state] then
24:                    root[prev] = min(root[prev], root[state])
25:                end if
26:                if requestStack.top() ≠ (VISIT, prev, any) then
27:                    if root[prev] = prev then
28:                        if fullyExpanded() then
29:                            repeat
30:                                w = componentStack.pop()
31:                                inComponent[w] = true
32:                            until w = prev
33:                        else
34:                            if (enabled = enabled(prev)) = null then
35:                                return false
36:                            end if
37:                            for all states succ ∈ enabled do
38:                                requestStack.push(VISIT, prev, succ)
39:                            end for
40:                            prev.fullyExpanded(true)
41:                        end if
42:                    end if
43:                end if
44:            end if
45:        end if
46:    end while
47:    return true
48: end function
```

Algorithm 4 Iterative Tarjan's Algorithm with Partial Order Reduction for Nonblocking

```
1: function IsNonBlocking(init)
2:     requestStack.push(VISIT, null, init)
3:     while requestStack ≠ ∅ do
4:         (req, prev, state) = requestStack.pop()
5:         if req = VISIT and state not visited then
6:             state.setVisited(true)
7:             root[state] = state
8:             if state has successors then
9:                 inComponent[state] = false
10:                componentStack.push(state)
11:                requestStack.push(CLOSE, prev, state)
12:                for all states succ ∈ ample(state) do
13:                    requestStack.push(VISIT, state, succ)
14:                end for
15:            else
16:                inComponent[state] = true
17:                if ¬isMarked(state) then
18:                    return false                          ▷ Deadlock detected
19:                end if
20:            end if
21:        else
22:            if prev ≠ null then
23:                if ¬inComponent[state] then
24:                    root[prev] = min(root[prev], root[state])
25:                end if
26:                if requestStack.top() ≠ (VISIT, prev, any) then
27:                    if root[prev] = prev then
28:                        if fullyExpanded() then
29:                            blocking = true
30:                            repeat
31:                                w = componentStack.pop()
32:                                inComponent[w] = true
33:                                if isMarked(w) then
34:                                    blocking = false
35:                                else
36:                                    if more than one component then
37:                                        if canReachComponent(w) then
38:                                            blocking = false
39:                                        end if
40:                                    end if
41:                                end if
42:                            until w = prev
43:                            if blocking then
44:                                return false              ▷ Livelock detected
45:                            end if
46:                        else
47:                            for all states succ ∈ enabled(prev) do
48:                                requestStack.push(VISIT, state, succ)
49:                            end for
50:                            prev.fullyExpanded(true)
51:                        end if
52:                    end if
53:                end if
54:            end if
55:        end if
56:    end while
57:    return true
58: end function
```

As was suggested, the nonblocking verifier is a bit more involved than the controllability verifier as it deals with reachability of states. We will now be referring to Algorithm 4. Every time we create a component we now must check for the conditions given in Definition 3.3. We see on line 18 this amount to simply checking if *state* is marked with a call to $isMarked(state)$. This is because at this point it has already been determined that *state* has no outgoing transitions, so it is clearly unable to reach any other strongly connected components. As such if that state itself is not marked then it violates the conditions given in Definition 3.3 and hence the model is blocking and we return false. The situation from line 29 is more interesting however. Now it must be verified if there exists a marked state in the strongly connected component, or if some state in the component can reach some other strongly connected component. To do this, as we create the component and each state w is popped from the *componentStack*, we first check if that state is marked with $isMarked(w)$, and if not then we check if it has a transition to some other component with $canReachComponent(w)$. In the latter case this check is only made if there has been more than one component created, as if there is only one component clearly there is no other component to reach. If either of these conditions is met then we set $blocking = false$, as the current component satisfies Definition 3.3. If this is not the case then we have detected a component that violates Definition 3.3, and hence we have found a livelock, and return false.

3.1.3 Ample set calculation

This section will introduce and detail the ample algorithm used in Algorithms 3 and 4. The ample algorithm developed during this research improves on the efficiency of the ample algorithm in previous work [17] shown in Algorithm 5 in a number of ways. It has been discussed that paying particular attention to how the events of the enabled set for a state are ordered can help to yield smaller ample sets on average, and those ordering will be explored here. Not only that but Algorithm 5 has a worst case run time complexity of $O(e^5)$ where e is the number of events in the model. This is due to the fact that

every time a new event is added to the current consideration of the ample set on line 8, all events are iterated over again in order to calculate any new dependencies on the ample set. The improved implementation shown in Algorithm 6 iterates over only the events that depend on each ample event instead of all events. As events are added to the end of the ample set, checking the dependencies of those events is deferred until the end of the ample iteration, so it is never necessary to restart the iteration. The worst case time complexity of this implementation of ample is $O(e^3)$.

The ample algorithm being introduced is designed so that the dependency condition **C2** is satisfied. As it stands, **C2** presents an enormously complex task to satisfy minimally, which would be the desired result. This is because it requires that no path from the current path may exist where any event that depends on ample occurs before an event in ample. To determine this precisely, multiple searches of the state space of the complete synchronous composition must be performed on every calculation of an ample set for a state, and this calculation must occur for every state in the reduced model. This amount of searching has severe time requirements, leading to a run time complexity $O(e^2 * t * s * s_R)$ where t is the number of transitions in the model, s is the number of states in the model, and s_R is the number of states in the reduced model. Because of this we introduce local conditions which are easier to verify and which satisfy **C2**.

C2.1 Every $\alpha \in ample(s)$ is independent of any $\beta \in enabled(s) \setminus ample(s)$.

C2.2 Let $D = \{\alpha \in \Sigma \setminus ample(s) \mid \alpha \text{ depends on some } \beta \in ample(s)\}$. Any $\alpha \in D$ cannot become enabled through the actions of some automaton M, using only events that are independent of every event $\beta \in ample(s)$.

The ample algorithm developed as part of this research is given in Algorithm 6 and we will use this to describe how the local conditions **C2.1** and **C2.2** have been satisfied. We begin by initialising four sets. The *ample* set is the set that will be used to contain the ample events of state s, the *dependentNotEnabled* set will be used to represent the set D defined in condition **C2.2**, the *dependent* set represents the union of sets *ample* and

Algorithm 5 Original ample set computation

```
1: function AMPLE(State s)
2:     ample = ∅
3:     dependent = ∅
4:     for all α ∈ enabled(s) do
5:         ample = {α}
6:         for all β ∈ Σ \ ample do
7:             if β depends on ample then
8:                 if β ∈ enabled(s) then
9:                     ample = ample ∪ {β}
10:                    dependent = ∅
11:                    restart loop in 6
12:                else
13:                    dependent = dependent ∪ {β}
14:                end if
15:            end if
16:        end for
17:        independent = Σ \ (ample ∪ dependent)
18:        for all σ ∈ dependent do
19:            danger = false
20:            for all M_i where σ ∈ Σ_i do
21:                if canBecomeEnabled(σ, M_i, independent, s_i) then
22:                    danger = true
23:                    break
24:                end if
25:            end for
26:            if danger = true then
27:                next α
28:            end if
29:        end for
30:        return ample
31:    end for
32:    return enabled(s)
33: end function
```

Algorithm 6 New ample set computation

```
1: function AMPLE(State s)
2:     ample = ∅
3:     dependentNotEnabled = ∅
4:     dependent = ∅
5:     enabled = enabled(s)
6:     while ¬enabled.isEmpty() do
7:         α = enabled.removeFirst()
8:         ample = ample ∪ {α}
9:         for all β ∈ ample do
10:            for all γ where γ depends on β do
11:                if γ ∈ enabled(s) then
12:                    ample = ample ∪ {γ}
13:                else
14:                    dependentNotEnabled = dependentNotEnabled ∪ {γ}
15:                end if
16:                dependent = dependent ∪ {γ}
17:            end for
18:        end for
19:        for all σ ∈ dependentNotEnabled do
20:            danger = false
21:            for all M_i where σ ∈ Σ_i do
22:                if canBecomeEnabled(σ, M_i, dependent, s_i) then
23:                    danger = true
24:                    break
25:                end if
26:            end for
27:            if danger = true then
28:                ample = ∅
29:                next α
30:            end if
31:        end for
32:        if ample.size() = enabled(s).size() then
33:            s.setFullyExpanded(true)
34:        end if
35:        return ample
36:    end while
37:    s.setFullyExpanded(true)
38:    return enabled(s)
39: end function
```

dependentNotEnabled, and *enabled* represents the enabled set of events of state s. We begin by initialising all of these sets but *enabled* as empty. The *enabled* set we initialise with a call to $enabled(s)$ which will return the enabled events of state s. Specific to this implementation is the ordering of the events that are returned by $enabled(s)$, so we choose our ample events by removing events from the beginning of *enabled*. We continue to remove items from *enabled* and consider them for the ample set until *enabled* is empty. To consider them for the ample set we take the event removed from *enabled* and add it to *ample*. The we iterate over all the events currently in *ample* and consider the events that depend on each ample event. We calculate ahead of time, when we determine the independencies, the set of dependent events for every event in the model. This means that we can immediately iterate over the dependent events without needing to calculate anything. Every dependent event γ falls into one of two categories; it is either enabled in the current state or not. If γ is enabled in the current state it is added to *ample* and if it is not enabled in the current state then it is added to *dependentNotEnabled*. In both cases it is also added to *dependent*. During this process of iterating over *ample*, *ample* grows by including all enabled events that are dependent on events in *ample*. We are guaranteed to satisfy **C2.1** once this is completed as the process will only terminate once either no events that are not already in *ample* are dependent on γ, or there are no events that depend on γ.

Once we have our candidate ample set we then go about satisfying **C2.2**, so we must determine that none of the events in *dependentNotEnabled* can become enabled in any of the component automata using only events from $\Sigma \backslash dependent$. To do this we iterate over the events of *dependentNotEnabled*. For each of these events σ we then iterate over each of the component automata M_i where $\sigma \in \Sigma_i$ and make a call to the algorithm given in Algorithm 7. Algorithm 7 determines the above requirement. It is improved slightly from the equivalent implementation given in [17] in that only the events local to the automaton that is being considered are used to determine if the dependent events can become enabled. This is done by creating the set *independent* and iterating over the local events of automaton M_i. If we have event $\alpha \in \Sigma_i$ and

$\alpha \notin$ *dependent* then we have identified α as one of the events that must be used to check the reachability of the dependent event σ in automaton M_i, so we add α to *independent*. Once *independent* contains all local events independent of ample, we then perform a depth first search of the state space of automaton M_i using only transitions that involve events from *independent*. If at any point in the search we encounter a state t_i such that $t_i \xrightarrow{\sigma}_i$ when then know that an event that depends on ample can become enabled using only events independent of ample in automaton M_i, so we return true. If we exhaust the depth first search and never encounter such a state then we have verified that σ can not become enabled, so we return false. Since this is checked for every component automata for every dependent event, if it was never the case that Algorithm 7 returned true then we can safely say that no events that depend on *ample* may become enabled using only events independent of *ample*. As such we have an ample set satisfying **C2.1** and **C2.2** and we propose that this ample set then satisfies condition **C2**, a proof of which is offered in Section 3.2. We can then return this ample set for use in the construction of the reduced model. We also know that as soon as Algorithm 7 returns true that this is not the case, so we begin the loop in line 6 again to consider a new ample set. Since the ample sets calculated in this way are in closure under the dependency relation, it is needless to consider new events that are a part of a previously computed ample set, as these events will generate the same ample set. Because of this we maintain another set *considered* that contains all events that have been included in any ample set as part of the current ample set calculation, and only events in *enabled* $\setminus$ *considered* are the considered for a new ample set.

3.1.4 Ordering of events

A large part of the optimisations developed during this research are achieved by an ordering of the events in order to try and calculate the smallest possible ample sets. We will explore two event orderings in this section; one based on the number of independencies of events, and one based on whether or not the inclusion of events would lead to the creation of more states. This

Algorithm 7 Determining enabledness of events in local automata

```
1: function CanBecomeEnabled(Event σ, Automaton M_i, Event[] dependent, State s_i)
2:     independent = ∅
3:     for all α ∈ Σ_i do
4:         if α ∉ dependent then
5:             independent = independent ∪ α
6:         end if
7:     end for
8:     stack = ∅
9:     stack.add(s_i)
10:    while stack ≠ ∅ do
11:        currentstate = stack.pop()
12:        if currentstate --σ-->_i then
13:            return true
14:        end if
15:        for all β ∈ others do
16:            if currentstate --β-->_i nextstate then
17:                stack.add(nextstate)
18:            end if
19:        end for
20:    end while
21:    return false
22: end function
```

section will introduce and detail the algorithms used to perform these event orderings.

The way in which these event orderings are proposed to result in a smaller state space in the reduced model is by selecting events from the enabled set of events for ample, such that the ample sets created by using those events are as small as possible. In order to achieve this we give an ordering to the enabled set of events, so that the ample algorithm can select the events appropriately. The way in which this ordering is determined is by adding the enabled events to a priority queue as they are discovered, where the weightings used to determine the priority for the events is determined based on the selection strategy that is being used.

Algorithm 8 shows the algorithm used to find the enabled events and order them based on whether or not the events are involved in a transition whose target state has already been visited. The majority of this algorithm remains unchanged from the similar algorithm given in [17]. What differs here is that *enabled* is now a priority queue, meaning that as the events are added to it they are now sorted with respect to the weighting given to them. To determine this weighting we must calculate the target state for each event considered. We create a state variable t to represent this target state, which is initially given the same value as the current state $s = (s_0, s_1, \ldots, s_n)$. We then consider each event in the model to determine their enabledness. For each event $\alpha \in \Sigma$ we then iterate over each component automaton M_i with $0 \leq i \leq n$. If it is not the case that α appears in the alphabet of M_i then we continue to the next automaton M_{i+1} as this means that the local state s_i is not affected by a transition on event α, giving us $t_i = s_i$ in this case. If however α is in the alphabet of M_i we then check to see if s_i enables α. If this is not the case then this means that an α transition may not occur from state s, so we have $s \overset{\alpha}{\not\rightarrow}$ and we can consider the next event. We must also consider the case of controllability in this case however, as if M_i is a specification and α is an uncontrollable event, then that means we have discovered an uncontrollable state. In this instance we can terminate

Algorithm 8 Enabled with previously visited ordering

```
1: function ENABLED(State s)
2:     enabled = new PriorityQueue
3:     t = s
4:     for all α ∈ Σ do
5:         for all M_i with 0 ≤ i ≤ n do
6:             if α ∉ Σ_i then
7:                 continue
8:             end if
9:             if s_i ↛^α then
10:                if M_i is specification ∧ α is uncontrollable then
11:                    terminate algorithm          ▷ System is uncontrollable
12:                else
13:                    next event
14:                end if
15:            else
16:                t_i = s'_i where s_i →^α s'_i
17:            end if
18:        end for
19:        if stateSet.contains(t) then
20:            setVisited(α)
21:        end if
22:        enabled.add(α)
23:    end for
24:    return enabled
25: end function
```

Algorithm 9 Event independency ordering algorithm

```
1: function DependencyWeightings
2:     for all event pairs (α, β) do
3:         if α depends on β then
4:             dependencyWeightings[α] = dependencyWeightings[α] + 1
5:             dependencyWeightings[β] = dependencyWeightings[β] + 1
6:         end if
7:     end for
8:     updateComparator(dependencyWeightings)
9: end function
```

the algorithm as we have determined that the model is uncontrollable. Note that this algorithm is still safe to be used for nonblocking verification as all automata in models used in nonblocking verification are set to be plants, so the check for specifications will always be false. If it was the case that s_i does enable α, then we store the target state of that transition to our target state t. Once all component automata have been iterated over, t represents the result of a successful transition from state s on event α. We can then check to determine whether or not t is a previously visited state by seeing if it appears in *stateSet*, the set of states containing all states that have be included in the reduced model. If t has already been visited then we record that as shown in line 20. What this does is affect the comparator for the priority queue signifying that α is to be given a higher weighting. When α is then added to the priority queue on line 22 it appears first in the ordering of enabled events. Events that have not been found to be previously visited will be added to the end of the priority queue, giving us that when all enabled events have been added in this way, we have our desired ordering.

Algorithm 9 shows the algorithm used to set the event weightings based on the number of independencies they share with other events. They way in which the events are selected for enabled is identical to how they are selected when using the selection strategy detailed above, except the target state no longer needs to be calculated as the weightings are calculated ahead of time. Since this is the case all enabled events may just be added to the priority queue *enabled* without any further calculation. This algo-

rithm is very simple as the majority of the work required for this is done in calculating the dependency relationship. That process remains identical to the algorithm given in [17]. Once the dependency relationship has been established we introduce an array of integers called $dependencyWeightings$, of size equal to the number of events in the model. This array is to store the number of events that each event depends on. For example if we have $dependencyWeightings[\alpha] = 5$ that would mean that event α depends on five other events. Now we consider each distinct pair of events and if those events depend upon one another, we increase the value for each of those events in $dependencyWeightings$ by one. By the time this has completed the values in the $dependencyWeightings$ array should contain the correct number of events that each event depends on. Now that this has been determined we can use this information in the comparator for the priority queue such that events with a fewer number of dependencies are given a higher priority than events with a higher number of dependencies, while events with identical numbers of dependencies are considered equal.

3.2 Proof of Correctness

This section provides proofs for the various conditions that have been asserted as sufficient in order to ensure that once the algorithms detailed in Section 3.1 have been performed, the resulting model will have the same properties of controllability and blocking as the original model.

Theorem 1 provides a proof that whenever the strict component condition **C4′** holds, then necessarily so too does the component condition **C4**. Theorems 2, 3, 4 and 5 together with Lemma 1 to provide a proof that as long as the ample conditions all hold, then the reduced model produced by the algorithms outlined in Section 3.1 with have the same properties of controllability and blocking as the original model. Theorem 6 provides a proof that Algorithm 6 outlined in Section 3.1 ensures that the ample dependency condition **C2** is satisfied. These three proofs together prove the correctness of the entire partial order reduction process outlined in this thesis with respect

to both controllability and nonblocking.

All of the proofs offered are based on existing work, however most of them have required some extra consideration so that they apply to this particular implementation of partial order reduction in discrete event systems for controllability and nonblocking. Theorem 1 is based on a proof taken from [3] adapted to work with strongly connected components, and shows that a weaker criterion for the component condition **C4** may be used while preserving the result. The proofs on pages 65 and 115 however have been altered somewhat from their sources [3, 6] in order to prove the correctness of the conditions with respect to the criteria mentioned above.

Since the partial order reduction process is now being used to verify nonblocking as well as controllability, we have generalised the proof offered in [6,17]. To do this we have identified the criteria by which the source proofs operated and created Lemma 1 that could then be used to prove that the partial order reduction process preseves both controllability and nonblocking. Theorem 2 together with Theorem 3 show that the process preserves nonblocking, and Theorem 4 together with Theorem 5 show that the process preserves controllability. The proof in Theorem refthm:prfamplealg differs from the original proof [3] as the way in which ample sets are chosen in [3] differ slightly from the way the algorithm given in Algorithm 6 describes. In the algorithm we have described, the ample sets for global states are not restricted to the enabled sets for particular local states as they are in [3], thus the proof that the algorithm still satisfies **C2** is altered to reflect the way ample sets are selected accordingly. The principle behind our proof however is similar to the one offered in [3].

3.2.1 The strict component condition is a sufficient criterion for the component condition

The result taken from [3] shows that whenever it is the case that one state on a cycle is fully expanded and the dependency condition **C2** holds for every

state on the cycle, then there can not be an event in $enabled(s_i)$ for some state s_i on the cycle, that is not included in $ample(s_j)$ for some other state s_j on the cycle. We have introduced a weaker condition that uses strongly connected components instead of cycles.

It was noticed that since each cycle belongs to a strongly connected component, then the same nature of reachability used in the proof for the so called *strict cycle condition* is also present when the matter is raised to the component level. The strict component condition then requires only a single state in each strongly connected component need be fully expanded for the result in **C4** to be achieved. The following offers a proof of this.

Theorem 1. If **C1** and **C2** hold, then **C4′** $\implies$ **C4**

Proof. By contradiction. Let $S' = \{s_0, s_1, \ldots, s_n\}$ be a component in a state graph produced using partial order reduction, where $ample(s_j) = enabled(s_j)$ for some $0 < j \leq n$, i.e., condition **C4′** is satisfied. Assume that **C4** is violated, i.e., for some $i \neq j$ there exists an event $\beta \in enabled(s_i)$ where $\beta \notin ample(s_k)$ for all $0 < k \leq n$. Consider the smallest path from state s_i to state s_j $s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \xrightarrow{\alpha_{i+2}} \cdots \xrightarrow{\alpha_j} s_j$. We know this path exists since S' is a strongly connected component and also from our assumption that $\alpha_l \neq \beta$ with $i \leq l \leq j$. Since this is a component in a reduced model, it follows that $\alpha_{i+1} \in ample(s_i)$ (only ample events are chosen when creating the reduced model). Condition **C2** yields that all events in $ample(s_i)$ are independent of all events in $enabled(s_i) \backslash ample(s_i)$. Since $\beta \in enabled(s_i) \backslash ample(s_i)$ it must be the case that α_{i+1} is independent of β. Since they are independent, it must also be the case that $\beta \in enabled(s_{i+1})$. From our assumption we know that $\beta \notin ample(s_l)$, meaning that β must still be enabled in every state s_l on that path. This must mean however that we have $\beta \in enabled(s_j) \setminus ample(s_j)$. This is of course a contradiction however since we asserted that state s_j was fully expanded. □

3.2.2 Ample conditions preserve properties of controllability and blocking

The following result is a modified version of the proof given in [6] which shows that as long as the three ample conditions **C1**, **C2** and **C3** are satisfied when choosing a reduced set of events for each state in an automaton, then the resulting reduced automaton will still satisfy the same property of controllability and nonblocking that the original automaton satisfied. The proof offered simplifies and generalises the original, encapsulating the core principle of the original proof in a lemma. The lemma is then used to prove that the ample conditions offered; including the revised condition **C3′**, retain properties of both blocking and controllability, instead of just controllability.

For the purposes of the following proofs, let $M = \langle \Sigma, S, S^\circ, \rightarrow, Q \rangle$ be a synchronous composition and let M_R be M produced by using partial order reduction.

Lemma 1. If $x \rightarrow y$ in M then there exists $y' \in S$ and $x \rightarrow y'$ in M_R with $y \rightarrow y'$ in M.

Proof. Let $C = x \rightarrow y$ be a path in M of length n.

It is shown by induction on $i = 0, \ldots, n$ that there exist paths η_i and θ_i such that $C = \eta_0 \circ \theta_0$ and

1. $\eta_i \circ \theta_i$ is a path in M
2. $\eta_i \circ \theta_i$ ends with a state reachable from y.
3. η_i starts at state x.
4. η_i is a path in M_R.
5. $|\theta_i| = n - i$.

Note that every state is considered reachable from itself, so condition1 is covered if $\eta_i \circ \theta_i$ ends with state y.

We are trying to obtain an path in M_R that starts at state x and ends with a state reachable from y, given that C is a path in M. Condition 5 shows that as i approaches n, η_i will constitute a larger and larger portion of the path, and when $i = n$ it will be the case that $\eta_i \circ \theta_i = \eta_i$. If conditions 1 and 2 both hold then it will be the case that $\eta_i \circ \theta_i$ is a valid path in M that ends with a state reachable from y. Condition 3 will ensure that the path begins at state x, and if condition 4 also holds then the η_i path will be a path in M_R. Together then the 5 conditions will yield the result of giving us a path in M_R beginning at x and ending in a state reachable from y, which is our desired result.

Base case $i = 0$. Let $\eta_0 = x$, $\theta_0 = C$. Clearly $\eta_0 \circ \theta_0 = x \circ C = C$

1. $\eta_0 \circ \theta_0 = C$ is a path in M by assumption.
2. $\eta_0 \circ \theta_0 = C$ ends with state y which is defined to be reachable from y.
3. $\eta_0 = x$ begins at state x.
4. $\eta_0 = x$ is a path in M_R as every state forms a path.
5. $|\theta_0| = |C| = n = n - 0$.

Assume that conditions 1-5 hold for i, now consider $i + 1$. We know that we have η_i and θ_i from the inductive assumption. Let $\theta_i = b_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} b_k$ where $k = n - i$, and let $|w|$ represent the length of the shortest path in M_R from w to a fully expanded state. We can safely do this as the component condition gives us that every component contains at least one fully expanded state. It can be shown by induction on $m = 0, \ldots, |b_0|$ that the paths η_{i+1} and θ_{i+1} exist.

Base case $m = 0$. In this instance it must be the case that b_0 is a fully expanded state, so $\alpha_1 \in \mathit{ample}(b_0)$. Let $\eta_{i+1} = \eta_i \circ b_0 \xrightarrow{\alpha_1} b_1$ and let $\theta_{i+1} = b_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_k} b_k$. Now $\eta_{i+1} \circ \theta_{i+1} = \eta_i \circ b_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} b_k = \eta_i \circ \theta_i$

1. $\eta_{i+1} \circ \theta_{i+1} = \eta_i \circ \theta_i$ is a path in M as this was part of the inductive assumption.

2. $\eta_{i+1} \circ \theta_{i+1} = \eta_i \circ \theta_i$ ends with a state reachable from y as this was part of the inductive assumption.

3. $\eta_{i+1} = \eta_i \circ b_0 \xrightarrow{\alpha_1} b_1$ has the same start state as η_i which is state x by inductive assumption.

4. $\eta_{i+1} = \eta_i \circ b_0 \xrightarrow{\alpha_1} b_1$ is a path in M_R since η_i is a path in M_R and $\alpha_1 \in ample(b_0)$ by assumption.

5. $|\theta_{i+1}| = |\theta_i| - 1 = n - i - 1 = n - (i + 1)$

Assume that for all η_i and $\theta_i = b_0 \to \cdots \to b_k$ that satisfy conditions 1-5 for m, there exist paths η_i' and θ_i' satisfying conditions 1-5 for $i + 1$. Show this is true for $m + 1$. We now have two cases:

- Some $\gamma \in ample(b_0)$ occurs on θ_i. Consider the first such event $\gamma = \alpha_j$. Ample condition **C2** gives us that all the events preceding γ on θ_i are independent of $ample(b_0)$. Therefore γ is independent of each α_l with $0 < l < j$, γ remains enabled in each state b_l. There then exist states c_l for every b_l where $b_l \xrightarrow{\gamma} c_l$. Again due to the independence of γ and each α_l there exist transitions $c_l \xrightarrow{\alpha_{l+1}} c_{l+1}$ for every c_l. This means that the paths $b_0 \xrightarrow{\alpha_1} b_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{j-1}} b_{j-1} \xrightarrow{\gamma} b_j$ and $b_0 \xrightarrow{\gamma} c_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{j-2}} c_{j-2} \xrightarrow{\alpha_{j-1}} b_j$ both exist in M. An example of this is shown in Figure 3.6.
  Now let $\eta_i' = \eta_i \circ b_0 \xrightarrow{\gamma} c_0$ and let $\theta_i' = c_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{j-2}} c_{j-2} \xrightarrow{\alpha_{j-1}} b_j \xrightarrow{\alpha_j+1} b_{j+1} \xrightarrow{\alpha_{j+2}} \cdots \xrightarrow{\alpha_k} b_k$. Then $\eta_i' \circ \theta_i' = \eta_i \circ b_0 \xrightarrow{\gamma} c_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{j-2}} c_{j-2} \xrightarrow{\alpha_{j-1}} b_j \xrightarrow{\alpha_j+1} \cdots \xrightarrow{\alpha_k} b_k$.

  1. $\eta_i' \circ \theta_i'$ is a path in M. It has been shown that the path $b_0 \xrightarrow{\gamma} c_0 \xrightarrow{\alpha_1} c_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{j-2}} c_{j-2} \xrightarrow{\alpha_{j-1}} b_j$ exists in M. The path $b_j \xrightarrow{\alpha_{j+1}} b_{j+1} \xrightarrow{\alpha_{j+2}} \cdots \xrightarrow{\alpha_k} b_k$ was already part of C which also exists in M. η_i is a path in M by inductive assumption. Since $\eta_i' \circ \theta_i'$ is a composition of paths that exist in M, it must also be a path in M.

2. $\eta_i' \circ \theta_i'$ ends with a state reachable from y since the end state b_k is the same as the end state of θ_i, which is reachable from y by inductive assumption.

3. $\eta_i' = \eta_i \circ b_0 \xrightarrow{\gamma} c_0$ has the same start state as η_i which is state x by inductive assumption.

4. $\eta_i' = \eta_i \circ b_0 \xrightarrow{\gamma} c_0$ is a path in M_R since η_i is a path in M_R and $\gamma \in ample(b_0)$ by assumption.

5. $|\theta_i'| = |\theta_i| - 1 = n - i - 1 = n - (i+1)$, since the γ transition that was a part of θ_i has been removed and is now a part of η_i'.

- All events on θ_i are not in $ample(b_0)$. They are then all independent of of $ample(b_0)$ by ample condition **C2**. Let $b_0 = d_0 \xrightarrow{\gamma_1} d_1 \xrightarrow{\gamma_2} \cdots \xrightarrow{\gamma_{m+1}} d_{m+1}$ be a shortest path from b_0 to a fully expanded state in M_R. This fully expanded state exists because $|b_0| = m+1$ by assumption. Using the same reasoning as before we know that there exists the path $d_1 = c_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} c_k$ as shown in Figure 3.7. Now let $\eta_i'' = \eta_i \circ b_0 \xrightarrow{\gamma_1} d_1 = c_0$ and $\theta_i'' = c_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_k} c_k$. Then $\eta_i'' \circ \theta_i'' = \eta_i \circ b_0 \xrightarrow{\gamma_1} c_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{k-1}} c_{k-1} \xrightarrow{\alpha_k} c_k$. Note here that $|\theta_i''| = |\theta_i|$, and that $|c_0| < |b_0|$ where c_0 is the initial state of θ_i'' and b_0 is the initial state of θ_i. Since $|c_0| < |b_0|$ this suggests that as m increases, the distance to a fully expanded state from the initial state of the new θ path decreases, and thus will eventually reach 0. At this point this initial state e_0 of our θ path θ_i''' will have $\alpha_1 \in ample(e_0)$ at which point we can construct paths $\eta_i' = \eta_i''' \circ e_0 \xrightarrow{\alpha_1} e_1$ and $\theta_i' = e_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_k} e_k$.

  1. $\eta_i' \circ \theta_i'$ is a path in M. It has been shown that η_i''' exists and the path $e_0 \xrightarrow{\alpha_1} e_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{k-1}} e_{k-1} \xrightarrow{\alpha_k} e_k$ exists in M.

  2. $\eta_i' \circ \theta_i'$ ends with a state reachable from y since the end state b_k of θ_i is reachable from y by inductive assumption, and the end state e_k has been shown to be reachable from b_k.

  3. η_i' has the same start state as η_i which is state x by inductive assumption.

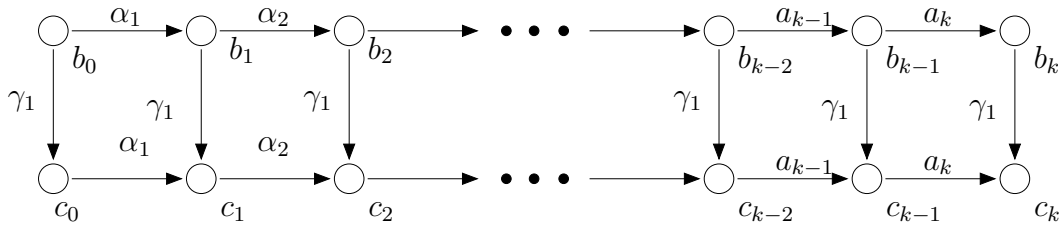

Figure 3.6: Example of independent events remaining enabled.

4. η_i' is a path in M_R since η_i''' is a path in M_R and $\alpha_1 \in ample(e_0)$ by assumption.
5. $|\theta_i'| = |\theta_i| - 1 = n - i - 1 = n - (i+1)$, since the α_1 transition that was a part of θ_i has been removed and is now a part of η_i'

□

Figure 3.6 shows how an event β independent of events $\alpha_0, \ldots, \alpha_n$, remains enabled in each successive state, and as such a parallel path $t_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} t_n$ exists. Figure 3.7 shows how successive independent events $\beta_0, \ldots, \beta_k$ give rise to a series of parallel paths $t_0^j \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} t_n^j$ with $0 \leq j \leq k$.

Theorem 2. Let M^ω be the abstraction of automaton $M = \langle \Sigma, S, S^\circ, \rightarrow, Q \rangle$ achieved using the process defined in Definition 2.15, then
M is nonblocking $\iff$ $\forall s \in S$ where $M^\omega \rightarrow s : s \xrightarrow{p\omega}$ for some path p, and
M is blocking $\iff$ $\exists s \in S$ where $M^\omega \rightarrow s : s \not\xrightarrow{p\omega}$ for all paths p.

Proof. Assume M is nonblocking, we must show that for all $s \in S$ where $M^\omega \rightarrow s$ we have $s \xrightarrow{p\omega}$. Consider an arbitrary state $s \in S$ where $M^\omega \xrightarrow{q} s$. Since M^ω is constructed by only adding ω transitions to $\perp$ we have $M \xrightarrow{q} s$. Since M is nonblocking we have $s \rightarrow t$ where $t \in Q$. We also have $t \xrightarrow{\omega} \perp$ in M^ω, and again since only ω transitions were added to construct M^ω we have $M^\omega \xrightarrow{q} s \rightarrow t \xrightarrow{\omega} \perp$. Since this was true for arbitrary reachable state s, we have for each reachable state $s \in S$, $s \xrightarrow{p\omega}$, which is our result.

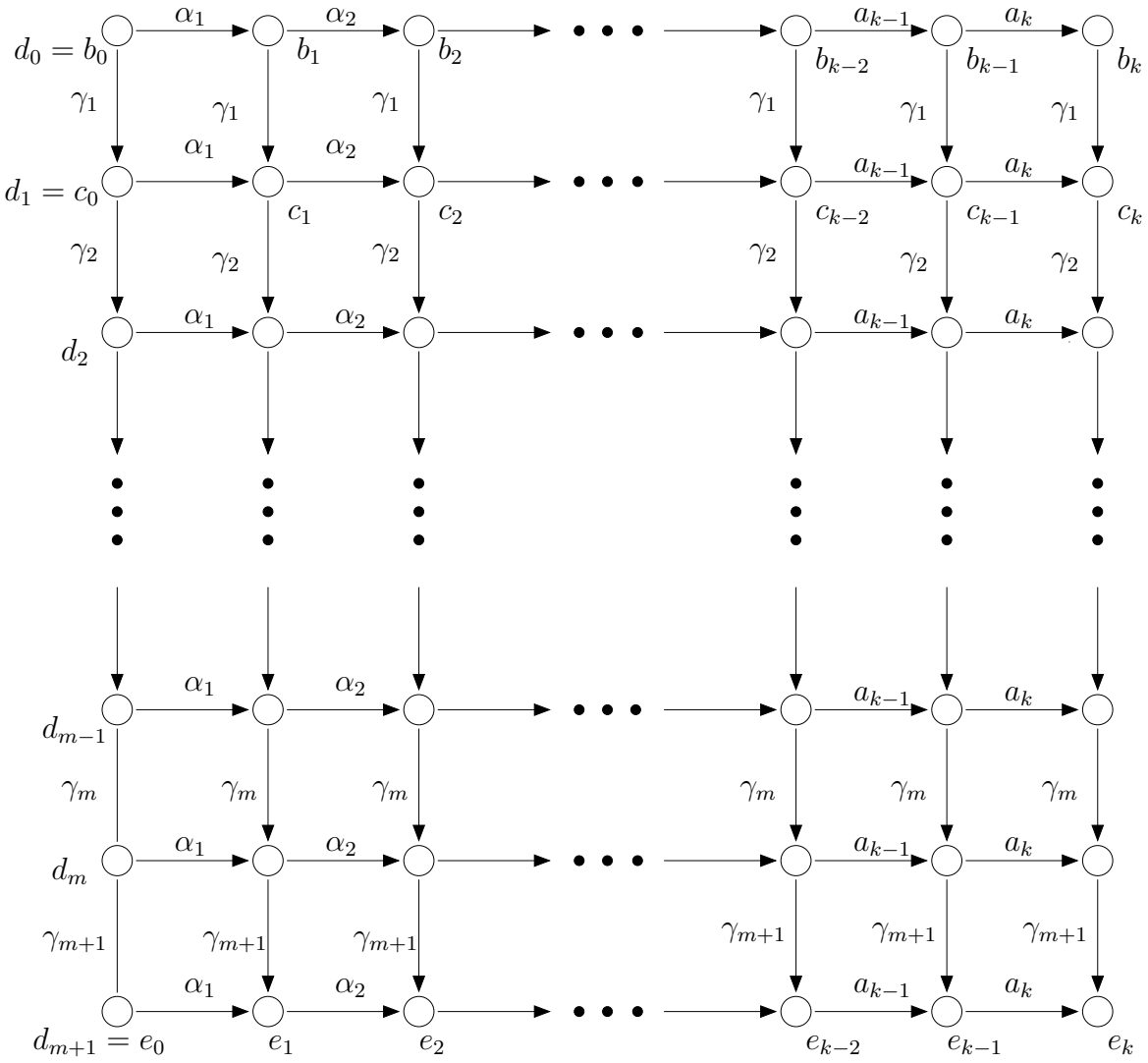

Figure 3.7: Example of events remaining enabled over a succession of independent events.

Now assume that for all $s \in S$ where $M^\omega \to s$ we have $s \xrightarrow{p\omega}$ for some path p. We must show that M is nonblocking. Consider an arbitrary state $s \in S$ where $M \to s$. Since M^ω is constructed by only adding ω transitions to $\perp$ we have $M^\omega \xrightarrow{q} s$. From our assumption we have $s \xrightarrow{p} t \xrightarrow{\omega}$ for some state t and path p. From the construction of M^ω we have $t \in Q$, and again since we only added ω transitions we have $s \xrightarrow{p} t$ in M, which gives us $M \to s \xrightarrow{p} t$. Since $t \in Q$ and this was true for arbitrary reachable state s, we have M is nonblocking.

Assume M is blocking, we must show that there exists a state $s \in S$ where $M^\omega \xrightarrow{p} s$ and for all paths p, $s \not\xrightarrow{p\omega}$. From the construction of M^ω and since M is blocking, we have $\forall s \in S : s \not\xrightarrow{\omega}$. This immediately gives us our result.

Now assume that there exists a state $s \in S$ where $M^\omega \xrightarrow{p} s$ and for all paths p, $s \not\xrightarrow{p\omega}$, we must show that M is blocking. Since M^ω is constructed by only adding ω transitions to $\perp$ we have $M \xrightarrow{q} s$. From our assumption there are no paths to any state t where $t \xrightarrow{\omega}$. From the construction of M^ω this means that there exist no paths to any state t where $t \in Q$, which gives us our result that M is blocking. □

Theorem 3. Let $M = \langle \Sigma, S, S^\circ, \to, Q \rangle$ then M is nonblocking $\implies$ M_R is nonblocking and M is blocking $\implies$ M_R is blocking.

Proof. From Theorem 2 this is equivalent to saying that if we have $s \to \perp$ for all reachable states s in M^ω, then the same is also true for all reachable states in M_R^ω, and if there exists a reachable state s in M^ω where $s \not\to \perp$ for all paths p, then there also exists such a state M_R^ω.

Assume M is nonblocking and let s be an arbitrary reachable state in M_R^ω. We have that s is also reachable in M^ω as M_R^ω is constructed only by removing transitions from M^ω and not adding them. Consider the path $s_0 \to \perp$ in M^ω which must exist since we know that s is co-reachable. By Lemma 2 we know that since $s \to \perp$ in M^ω then $s \to \perp'$ in M_R^ω where $\perp \to \perp'$. Since the only transitions from $\perp$ are selfloops however this means

that $\perp' = \perp$, which gives us $s \to \perp$ in M_R^ω. Since the only transitions to $\perp$ are on event ω, we have $M_R^\omega \to s \to t \xrightarrow{\omega} \perp$ for some state $t \in S$. From the construction of M^ω we have $t \in Q$, therefore an accepting state is reachable from s_0 in M_R, giving us our result that any arbitrary reachable state of M_R is also co-reachable and hence M_R is non-blocking.

If M is blocking then there exists a state $s \in S$ where $M^\omega \to s$ such that $s \not\to \perp$. It follows that it must be the case that any state $s' \not\to \perp$ where $s \to s'$. By Lemma 2 we know that such a state s' is reachable in M_R^ω, giving us our result that a non-coreachable state is reachable in M_R^ω and hence M_R is also blocking. □

Theorem 4. Let $M' = \langle \Sigma, S', S^\circ, \to', Q \rangle$ be the abstraction of automaton $M = \langle \Sigma, S, S^\circ, \to, Q \rangle = M_1 || \ldots || M_n$ achieved using the process defined in Definition 2.13, then
M is uncontrollable $\iff$ $\exists s = (x_0 \ldots, x_n) \in S'$ where $x_i = \perp$ for some $0 \leq i \leq n$.

Proof. Assume M is uncontrollable, we must show that $\exists s = (x_0 \ldots, x_n) \in S'$ where $x_i = \perp$ for some $0 \leq i \leq n$. Since M is uncontrollable there exists a state reachable $t = (y_0, \ldots, y_n) \in S$ where for all plant automata M_j we have $y_j \xrightarrow{\alpha} x_j$, and for some specification automaton M_i we have $y_i \not\xrightarrow{\alpha}$ with $\alpha \in \Sigma_i \cap \Sigma_u$. Since plantification only adds transitions on uncontrollable events to the dump state $\perp$, we have $S \subseteq S'$, and so $S' \to s$. From the construction of M' we have $y_i \xrightarrow{\alpha} x_i = \perp$, which in turn yields the transition $t = (y_0, \ldots, y_n) \xrightarrow{\alpha}{}' s = (x_0, \ldots, x_i = \perp, \ldots, x_n)$, which is our result.

Assume $\exists s = (x_0 \ldots, x_n) \in S'$ where $x_i = \perp$ for some $0 \leq i \leq n$, we must show that M is uncontrollable. By contradiction, assume M is controllable. This means that for all specification automata M_j, $\alpha \in \Sigma_j \cap \Sigma_u \implies \forall t \in S_j : t \xrightarrow{\alpha}$. From the construction of M' this means that there exist no transitions to the dump state $\perp$. Since $x_i = \perp$ however this means that such a transition must have occurred, so we have a contradiction and therefore M is uncontrollable. □

Theorem 5. M is controllable $\implies$ M_R is controllable and M is uncontrollable $\implies$ M_R is uncontrollable.

Proof. If M is controllable then every state of M is a controllable state. Since states are only ever removed and not added when constructing M_R it must be the case then that M_R also contains only controllable states and hence is controllable.

If M is uncontrollable then there exists some counterexample $s_0 \to s_n$ in M where s_n is an uncontrollable state. By Theorem 4, this means that for some x_k with $s_n = (x_0, \ldots, x_m)$ it is the case that $x_k = \perp$. Since $\perp$ has selfloops for all events, it is the case that $x_k \xrightarrow{\alpha} x_k$ for all $\alpha \in \Sigma$. This means that any state s'_n where $s_n \to s'_n$ must also be an uncontrollable state as it includes $x_k = \perp$. By Lemma 2 we know that such a state s'_n is reachable in M_R, giving us our result that an uncontrollable state is reachable in M_R and hence M_R is also uncontrollable. □

3.2.3 Ample algorithm satisfies the dependency condition

The following result will use a modified version of the proof offered in [3] to show that the steps taken by the ample algorithm implemented in this thesis satisfy the dependency condition **C2**. That is to say that as long as **C2.1** and **C2.2** hold in state s, then no sequence of events taken from state s will result in an event that is dependent on some event $\alpha \in \mathit{ample}(s)$ before some event $\beta \in \mathit{ample}(s)$ has occurred, where it may be the case that $\beta = \alpha$.

Theorem 6. If **C2.1** and **C2.2** hold in state s, then $\mathit{ample}(s)$ will satisfy **C2** for all sequences of transitions from state s.

Proof. By contradiction. Let $M = \langle \Sigma, S, \to, S^\circ \rangle$ be a synchronous composition and let $s_0 = (x_0, x_1, \ldots, x_n) \in S$. To select $\mathit{ample}(s_0)$, first some event $\alpha \in \mathit{enabled}(s_0)$ is considered and added to $\mathit{ample}(s_0)$, then all of the events that depend on α that are also in $\mathit{enabled}(s_0)$ are included in $\mathit{ample}(s_0)$ as well. All of the enabled events that depend on the events added in this way

are then also included in $ample(s_0)$, this repeats until no more events are added. The set $ample(s_0)$ is then comprised entirely of dependent enabled events, which leaves two other sets of events to consider; events that are independent from all the events of $ample(s_0)$, call this set I, and events not in $enabled(s_0)$ that are dependent on one or more of the events in $ample(s_0)$, call this set D. Recall that $ample(s_0)$ is only considered viable by the algorithm presented in Chapter 3 if there exists no component automaton M_i where an event from D can become enabled in the final state of a path from state x_i using only events from I.

Assume then that ample condition **C2** is violated after $ample(s_0)$ has been created in this way. Then there exists some path from state s_0 where an event dependent on an event in $ample(s_0)$ occurs before an event from $ample(s_0)$. Let this path be $s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_j} s_j$, where β_j depends on an event in $ample(s_0)$ and each β_i with $0 \leq i < j$ is independent of all of the events in $ample(s_0)$, hence each $\beta_i \in I$. Since all of the dependent enabled events for state s_0 were included in $ample(s_0)$ it must be the case that $\beta_j \in D$. Consider the component automata $M_0, M_1, \ldots, M_n$ in states $s_0, \ldots, s_{j-1}$. Since $\beta_j \in enabled(s_{j-1})$, this means that each M_k where $\beta_j \in \Sigma_k$ has transitioned to a state x'_k where $\beta_j \in enabled(x'_k)$. There must exist at least one such M_k, as if was the case that $\beta_j \notin \Sigma_k$ for all $0 \leq k \leq n$, then it would be the case that $\beta_j \notin \Sigma$, which is false by assumption. Also it must have been the case that $x_k \xrightarrow{\beta_j}\!\!\!\!\!\!/\;$ for some k where $\beta_j \in \Sigma_k$, as otherwise it would be the case that $\beta_j \in enabled(s_0)$, contradicting the fact that $\beta_j \in D$. This M_k has then transitioned to this state x'_k, however only events exclusively from I have been used, meaning that $\beta_j \in D$ has become enabled in automaton M_k using only events from I, which contradicts how the algorithm chose $ample(s)$. Therefore our assumption that **C2** has been violated must be incorrect, so the algorithm must satisfy **C2**. □

3.3 Experimental Results

The algorithms outlined in Section 3.1 have been implemented in the WATERS [1, 18] model checking suite and have been run on various models to determine how much reduction can be observed. Another metric of significance will be the time taken to run the algorithm on models of various sizes. When compared to a standard model verifiers that create the entire state-space, much more computation is done for each state explored in a partial order reduction verifier. This is because on top of calculating the enabled set of events for each state, the partial order reduction process requires using the enabled set to compute an ample set. Most the time spent in the partial order reduction process is spent calculating ample sets, so this would suggest that the algorithm would take longer to perform the verification in all cases. Since a smaller set of successor states are being explored in some cases however, this reduces the number of states for which this computation is being performed. Because of this there should exist some relationship between how effective the partial order reduction method is at reducing the state space of a given model and the run time for that model. Tables 3.1, 3.2 and 3.3 shows the results of analyzing various models for with the different algorithms developed in Section 3.1.

Table 3.1 compares using a regular monolithic model checker, the partial order reduction verifier given in [17] which makes use of the ample algorithm given in Algorithm 5, and a partial order reduction verifier that has been implemented using the optimised ample algorithm given in Algorithm 6. The monolithic model checker explores every successor of every state using a breadth-first search. The two partial order reduction verifiers used in this table differ only by the ample algorithm used. This means that they are both using the depth first search given in [17], which uses the cycle condition **C3** instead of the component condition **C4** developed in this chapter. There are five metrics considered in this table, given by the five columns States, Time, Cycles, Reduced Sets, and Full Expansions. States and Time should be self explanatory, so only the remaining three metrics will be explained. The *Cy-*

cles metric gives the number of times that a partial order reduction algorithm generated a transition that closed a cycle. This is of importance since the depth first search algorithm being used uses the fact that a cycle was closed in order to know when to fully expand states, thereby potentially increasing the state-space. It is therefore advantageous to try have as few cycles closed as possible. The *Reduced Sets* metric gives the number of times an ample set was created for a state by a partial order verifier, that was smaller than the enabled set for that state. Obviously the greater this value is the better, as generating smaller ample sets is the main factor in reducing the state-space. Finally the *Full Expansions* metric gives the number of times that a partial order algorithm was forced to fully expand a state in order to comply with the cycle condition **C3**. A smaller value here is better as fully expanding states reduces the number of reduced sets in the process. It should be noted here that the values in the states column for the monolithic checker that are prefixed with a * represent models for which the monolithic checker could not construct the entire state space before running out of memory, as such the time spent for these models with the monolithic checker is undetermined and so is represented by a ? in the time column. In these cases, other techniques have been used to discover the number of states in these large models. The three columns Aut, Events and Cont under Model represent the number of component automata in the model, the number of events in the model, and whether or not the model is controllable, respectively.

Table 3.2 compares using a partial order reduction controllability verifier using the different selection strategies given in Algorithms 8 and 9. For each of these comparisons the depth first search algorithm given in Algorithm 3 is used, which means that for these tests the component condition **C4** is being used as opposed to the cycle condition **C3**. Also the ample set calculation is done with Algorithm 6, as we can see from Table 3.1 that this algorithm performs better than version given in [17] in almost all cases. The columns *PO Comp Ind*, *PO Comp Visit* and *PO Comp Both* give the results for running the verifiers in this way so that they differ only by the selection strategy used. The *PO Comp Ind* selection strategy uses the algorithm given in Algorithm

9 where events are ordered by the number of independencies they share with other events. The *PO Comp Visit* selection strategy uses the algorithm given in Algorithm 8 where events are ordered depending on whether or not they are involved in transitions to states that are already visited. Finally the *PO Comp Both* selection strategy uses both of the pervious strategies combined. The way that this works is that events with higher visited priority trump those events with a lower visited priority, and if events have the same visited priority then the independence priority is used. The metrics in this table remain the same, except the cycles metric has been replaced by the *Components* metric. Since Algorithm 3 calculates precisely all of the components of a model instead of estimating, as was the case for cycles in the algorithms given in [17], this information can be included under the model column. From the strict component condition **C4′**, we must only fully expand one state per strongly connected component, so we would expect fewer full expansions for these tests.

Table 3.3 compares using a partial order reduction nonblocking verifier with a monolithic nonblocking verifier. Again the monolithic verifier generates the entire state-space of the synchronous composition, only this time it is searching for blocking states. The partial order algorithm used is Algorithm 4. As with the previous set of tests this uses the component condition **C4**. It is no longer optional in this case, as we have from Definition 3.3 that components are used in order to determine in a model is nonblocking. This is also using the ample algorithm given in Algorithm 6 and the selection strategy given in Algorithm 9 for the best possible results. Included as an added metric in this case is *Reduction*. This gives the percentage reduction in state space achieved by partial order reduction over the standard monolithic implementation.

The models in this suite of tests can be grouped into four distinct groups; transferLine [4,21], central locking [15], philosophers [7], and profisafe [12,13].

The transferLine tests are based on systems created by the process outlined in [4]. This involves taking functional blocks, and combining them into larger systems which are effectively all of the blocks running in parallel. The blocks themselves are simple systems similar to the example given in Figure 2.5. Each consists of a machine, two buffers and a test unit. The first buffer *B1* stores units of work to pass to the machine, while the other buffer *B2* receives units of work from the machine and passes them on to the test unit. The test unit can either pass the work on to the next functional block, or send the work piece back to *B1* to be processed again by the machine. The capacities of *B1* and *B2* are 3 and 1 respectively. The tests model this system with 4 and 5 funtional blocks corresponding to the number in the model name for the test. It can be seen immediately that the state space of the full synchronous product increases drastically as more functional blocks are added, evidenced by the states for the monolithic model verifier in Table 3.1. Looking at the partial order reduction results we can see that PO old using Algorithm 5 barely achieves any reduction and takes significantly longer than the monolithic verifier. The PO new test using Algorithm 6 however achieves roughly a 20% reduction in of the full state-space and performs much faster then the older version, although not enough reduction is achieved in order to yield times lower than that of the monolithic verifier. Looking at the different selection strategies in Table 3.2 we see that none of them have managed to improve on the results of PO new. They all share the same state-space and differ only in the time taken to run. This suggests that there is not sufficient variation in the different ample sets that can potentially be computed and as such the improved ample algorithm suffices to find an effective ample set. Examining the nonblocking verifier results in table 3.3 however gives drastically different results. Here we see state-space reductions of 94.3% and 98.1% for transferline 4 and 5 respectively. The reductions are so great using the partial order reduction nonblocking verifier that two additional models with 6 and 7 functional blocks have been included with projected state spaces included for the monolithic nonblocking verifier. The reduction for transferline7 approaches 100% due the exponential rate at which the total state-space grows while the reduced model state-space only

grows linearly. Thanks to this extreme reduction in state-space these models are able to be verified very quickly as there are much fewer states to explore. The reason we observe such drastic improvements with the nonblocking verifier as opposed to the controllability verifiers we propose is due to how the dependency relation is affected in the two cases. In the controllability verifier the transition relation of the model is changed in order to construct the plantification of specification automata given in Definition 2.13. This adds transitions on existing uncontrollable events to the dump state $\perp$ that would not otherwise be present, forcing a dependency relationship between events. It is these dependencies that ensure the verifier will not exclude any uncontrollable states when computing ample sets. In the case of the nonblocking verifier however the abstraction defined in Definition 2.15 is used, which adds an ω transition to the dump state $\perp$ for each accepting state. This does not add any dependencies as ω is independent of every other event, due to the selfloops for all events on $\perp$. When considering how the system operates then we can hypothesise why reductions of this magnitude are observed. Essentially the functional blocks within the complete system are independent systems running in parallel. It is only when a unit of work is passed between the functional blocks that they share events. As a result of this we can see that the vast majority of the time there is only some number of independent events occurring, meaning that the vast majority of states in the system have transitions enabled largely on independent events. As this is the main criteria by which ample sets are selected, the result is that minimal selections for ample sets can be made most of the time, as we can see by the size of the reduced sets when compared to the number of states produced.

The central locking tests consist of dreitueren, koordwsp and koordwsp_block. These tests are part of the BMW central locking system modelled in the KorSys project [15]. As the description suggests they model the central locking system of automobiles. While this may seem like a simple task to model it can be seen from the state spaces from the complete synchronous product that these models are sufficiently complex. Again when verified for controllability we see similar results to that of transferline. When using the

old ample algorithm no reduction is achieved except for around 20% for dreitueren, and when the new ample algorithm is used we see reductions of around 50% for the koordwsp models while the results for the different selection strategies continue to remain consistent with those of PO new. We can put this down to the lack of varied potential ample set solutions again. In the nonblocking verifier case however we again see improved results. While dreitueren remains at around 20% reduction, koordwsp and koordwsp_block enjoy 77.1% and 99.5% reduction respectively. We can most likely discount the koordwsp_block result due to the fact that it is blocking, which means that it may just be luck that the depth first search of the partial order reduction verifier managed to find close a component satisfying the nonblocking check given in Definition 3.3 sooner than the breadth first search of the monolithic verifier. The high reduction achieved in the koordwsp model is likely due to the same reason as the high rates of reduction that were explained in the transferline case.

The philosoper tests are based on the classic computer science problem of the "dining philosophers" introduced by [7]. The problem is described as a number of philosophers sitting at a round table each with a bowl of food in front of them. In between each pair of adjacent philosophers is a fork. At each step, each philosopher may either pick up a fork, put down a fork, consume the food, or wait/think. Each philosopher may only eat if they are holding two forks. The problem is to devise a strategy that each philosopher can employ such that no philosopher can go forever without comsuming any food. Clearly a problem arises when the strategy of each philosopher is to first pick up the fork on his/her left, and then the fork on the right, and then eat. As soon as each philosopher picks up the left fork, no forks remain, and so each philosopher is waiting forever to pick up the right fork. This is known as deadlock. The dirty_philosopher and dining_philosopher tests have no controlling strategy and as such simply map the state space for all possible sequences of decisions any of the philosophers can make. The ordered philosophers tests add a strategy where deadlock is avoided by requesting forks in a fixed order. In all tests the number in

the model name denotes the number of philosophers sitting at the table. Clearly as the number of philosophers increases the state space very quickly grows as the permutation of events for each philosopher grows exponentially. These tests also exhibit quite large reduction, with ordered_philosophers_12 being unable to fit in memory when checked by the monolithic model checker while the partial order methods enjoy around a 70% state-space reduction. Interestingly, for the dining and ordered philosophers models the old ample algorithm achieves slightly more reduction than the new ample algorithm, although the new ample algorithm is substantially faster. To determine why this is we must examine the other three metrics, cycles, reduced sets, and full expansions. When we compare the cycles metric of the two implementations we see that the new implementation has closed substantially more cycles than the old implementation. There is no safeguard in either implementation that attempts to prevent this happening, and the random ordering of events allows for this to happen. This then impacts the full expansions metric and the new implementation is forced to fully expand more states. It may be reasonable to think then that the reduced sets metric would be lower for the new implementation, but we observe a higher number of reduced sets. This is due to the fact that while the new implementation may originally create more reduced ample sets, when a cycle is closed one of the states that achieved one of these reduced sets may very well then get fully expanded, causing the actual number of reduced sets to be lower than that which is reported. The results for the dirty philosopher models support this reasoning. These models are slightly better for the new implementation, and when we look at the cycles metric we see that for these models the new implementation creates fewer cycles than the old implementation. Looking at the results for the different selection strategies this also reinforces this reasoning. Here we begin to see further reductions in the state space once some ordering is given to the events chosen for the ample sets. These tests are also using strongly connected components in order to determine when full expansions have to happen which, when compared to the number of cycles and full expansions of the previous implementations, makes a large difference. We do still see however that it does not matter which strategy is used, with each

one differing only in the time taken to complete, which does still suggest that the set of different potential ample sets is not too great. Even more interesting are the results of the nonblockin verifiers. Again we can discount the dirty and dining philosopher models for the same reasons as we did for koordwsp_block as they are blocking, although the results do suggest that using strongly connected components to determine nonblocking could be a better approach than the approach used in the monolithic verifier, as the blocking states appear to be found much sooner in all cases. The nonblocking verifier however does not achieve as much reduction and the controllability verifiers for the ordered_philosopher models however, to the point where the ordered_philosohers_12 model could not be included in Table 3.3 due to the state-space being too large. As of the writing of this thesis an explanation of why this should be the case is not yet understood, and further analysis of this case may be required to further understand where the shortcomings and advantages of the partial order reduction process lie. In all cases however a large reduction in the total state-space is still achieved. This is similar to the case of the transferline example. The vast majority of the events are independent and most of the time, each philosopher is only concerned with his own actions and those of his neighbours.

The profisafe tests are based on the field bus protocol models introduced in [12, 13]. These tests serve in this case to illustrate that partial order reduction is not effective in all cases. Immediately we can observe that none of the partial order reduction methods have yielded any reduction whatsoever in any of these tests, while the tests perform slower than the monolithic checkers in all cases. This is to be expected of course as we have discussed earlier that the only time we will observe gains in run time will be when large gains in state space reduction are achieved. We do notice however that the new ample algorithm performs much faster than the old implementation. This is due mainly to the fact that the profisafe models have a staggeringly large number of events. As was discussed in section 3.1, Algorithm 5 has a worst case run time complexity of $O(e^5)$ where e is the number of events in the model. The optimisations made in Algorithm 6 are shown by these

results to have a significant impact on run time when verifying models with a large number of events such as these. It was initially hypothesised in [17] that part of the reason for the absence of any reduction was due to the large number of cycles, and hence fully expanded states, in the models. This has been shown to be false upon further analysis of these models however. As is somewhat reflected by the reduced sets metric in these results, the dependency relation of the events does not allow for the creation of reduced ample sets due to the dependency condition **C2**. For almost all ample set calculations the dependency relation is such that the entire enabled set is added to ample.

Figure 3.8 gives a chart showing state-space comparisons for most of the models when verified by the various model checkers outlined in this section. The models ordered_philosophers_12, transferLine_4, and tictactoe have been excluded fro the chart due to visibility issues brought on by scaling.

Model				States			Time(s)			Cycles		Reduced Sets		Full Expansions	
Name	Aut	Events	Cont.	Mono	PO old	PO new	Mono	PO old	PO new	PO old	PO new	PO old	PO new	PO old	PO new
transferLine_4	21	22	Yes	87578	86072	69603	0.253	1.062	0.450	108244	69051	15995	16825	22912	15718
transferLine_5	26	27	Yes	1280020	1268814	1017290	3.754	19.756	6.865	1868791	1428079	200045	245912	331793	244176
tictactoe	28	35	No	2422	43	43	0.058	0.020	0.018	0	0	0	0	0	0
dreitueren	33	74	Yes	420283	341561	341274	3.500	8.224	4.387	237768	283749	89623	98848	129041	140864
koordwsp	25	52	Yes	465648	465648	208996	3.020	23.824	3.784	2335120	766989	0	60652	452119	145609
koordwsp_block	24	42	Yes	634608	634608	327172	3.878	32.655	5.755	3170996	1198536	0	94348	616124	228638
ordered_philosophers_10	20	50	Yes	983038	422584	428889	3.451	6.846	2.143	140301	210954	335077	341382	36689	43515
ordered_philosophers_11	22	55	Yes	3932158	1451400	1502751	17.071	30.113	8.019	480308	723414	1175954	1227305	114229	138416
ordered_philosophers_12	24	60	Yes	*15728638	4938155	5094767	?	121.597	33.116	1594116	2192058	4071096	4227708	353081	415790
dining_philosophers_9	18	45	Yes	855093	682639	689182	2.573	8.945	3.370	289159	418477	420352	426895	89599	96582
dining_philosophers_10	20	50	Yes	3900559	2928081	2957325	13.760	45.166	16.551	1190092	1736750	1879326	1908570	344883	374146
dirty_philosophers_8	24	40	Yes	390623	229057	226749	1.346	3.435	1.226	112606	105625	191305	188997	49165	48529
dirty_philosophers_9	27	45	Yes	1953123	975819	966558	8.038	17.169	5.565	489781	470425	835216	825955	194257	189739
dirty_philosophers_10	30	50	Yes	*89765623	4108435	4102333	?	89.840	28.496	2089061	2026719	3584073	3577971	757463	740129
profisafe_i3host_efa	21	248	Yes	258056	258056	258056	3.524	65.654	11.512	609875	610138	0	0	105030	105038
profisafe_i4host_efa	21	298	Yes	508780	508780	508780	9.850	147.968	28.878	2335129	1215523	0	0	452119	210667
med_bmw	25	54	Yes	948024	923808	850139	4.492	21.588	9.290	1611779	1411569	59812	353755	244193	255802

Table 3.1: Table showing results of controllability checking using monolithic, and partial order reduction with both old and new ample implementations.

Model					PO Comp Ind				PO Comp Visit				PO Comp Both			
Name	Aut	Events	Cont.	Components	States	Time(s)	Reduced Sets	Full Expansions	States	Time(s)	Reduced Sets	Full Expansions	States	Time(s)	Reduced Sets	Full Expansions
transferLine_4	21	22	Yes	2	69603	0.459	16825	0	69603	0.527	16825	0	69603	0.525	16825	0
transferLine_5	26	27	Yes	2	1017287	7.337	245909	0	1017287	8.686	245909	0	1017287	8.543	245909	0
tictactoe	28	35	No	0	36	0	0	0	36	0.019	0	0	36	0.018	0	0
dreitueren	33	74	Yes	2	340641	3.598	91172	0	340799	4.207	90881	0	340791	4.199	90928	0
koordwsp	25	52	Yes	32	208725	2.719	60381	0	208725	3.428	60381	0	208725	3.132	60381	0
koordwsp_block	24	42	Yes	1183	326901	5.142	94077	0	326901	6.118	94077	0	326901	5.733	94077	0
ordered_philosophers_10	20	50	Yes	1	398573	2.102	311066	0	398573	2.471	311066	0	398573	2.495	311066	0
ordered_philosophers_11	22	55	Yes	1	1352598	8.107	1077152	0	1352598	9.799	1077152	0	1352598	9.696	1077152	0
ordered_philosophers_12	24	60	Yes	1	4566108	33.669	3699049	0	4566108	40.863	3699049	0	4566108	40.877	3699049	0
dining_philosophers_9	18	45	Yes	3	655353	3.703	393066	0	655353	4.387	393066	0	655353	4.432	393066	0
dining_philosophers_10	20	50	Yes	3	2796195	20.211	1747440	0	2796195	23.882	1747440	0	2796195	23.678	1747440	0
dirty_philosophers_8	24	40	Yes	203	148032	0.769	110280	0	148032	0.882	110280	0	148032	0.894	110280	0
dirty_philosophers_9	27	45	Yes	400	603528	3.207	462925	0	603528	3.744	462925	0	603528	3.748	462925	0
dirty_philosophers_10	30	50	Yes	790	2443569	15.098	1919207	0	2443569	17.956	1919207	0	2443569	18.143	1919207	0
profisafe_i3host_efa	21	248	Yes	22165	258056	7.998	0	0	258056	9.278	0	0	258056	9.147	0	0
profisafe_i4host_efa	21	298	Yes	30283	508780	19.728	0	0	508780	22.531	0	0	508780	22.433	0	0
med_bmw	25	54	Yes	5	824466	7.858	328082	0	824466	9.199	328082	0	824466	9.131	328082	0

Table 3.2: Table showing results of partial order reduction controllability verification using strongly connected components for the depth first search together with different selection strategies.

Model				States		Time(s)		Reduction	Reduced Sets	Full Expansions
Name	Aut	Events	Conf.	Mono	PO	Mono	PO			
transferLine_4	21	22	Yes	87578	4977	0.427	0.121	94.3%	4524	0
transferLine_5	26	27	Yes	1280020	24497	6.708	0.227	98.1%	22699	0
transferLine_6	31	32	Yes	*6584988	116353	?	0.688	98.2%	109178	0
transferLine_7	36	37	Yes	*273438928	538881	?	3.186	99.8%	510201	0
tictactoe	28	35	Yes	6324	6796	0.100	0.182	0%	6324	0
dreitueren	33	74	Yes	420283	340642	3.982	4.016	19.9%	91172	0
koordwsp	25	52	Yes	465648	106714	3.883	1.542	77.1%	46779	0
koordwsp_block	24	42	No	634608	3064	5.257	0.198	99.5%	646	0
ordered_philosophers_10	20	50	Yes	983038	600751	5.458	6.778	38.9%	414327	0
ordered_philosophers_11	22	55	Yes	3932158	2182911	27.159	30.036	44.5%	1546439	0
dining_philosophers_9	18	45	No	97464	6622	0.374	0.181	93.2%	826	0
dining_philosophers_10	20	50	No	382614	17305	1.427	0.283	95.5%	2850	0
dirty_philosophers_8	24	40	No	28725	1059	0.170	0.053	96.3%	69	0
dirty_philosophers_9	27	45	No	115806	1207	0.521	0.091	99.0%	43	0
dirty_philosophers_10	30	50	No	467274	7085	2.264	0.218	98.5%	437	0
profisafe_i3host_efa	21	248	Yes	258056	258061	3.958	8.510	0%	4	0
profisafe_i4host_efa	21	298	Yes	508780	508785	9.591	20.319	0%	4	0
med_bmw	25	54	Yes	948024	666766	6.829	7.179	29.7%	298132	0

Table 3.3: Table showing results of nonblocking verification using both monolithic and partial order reduction algorithm verifiers.

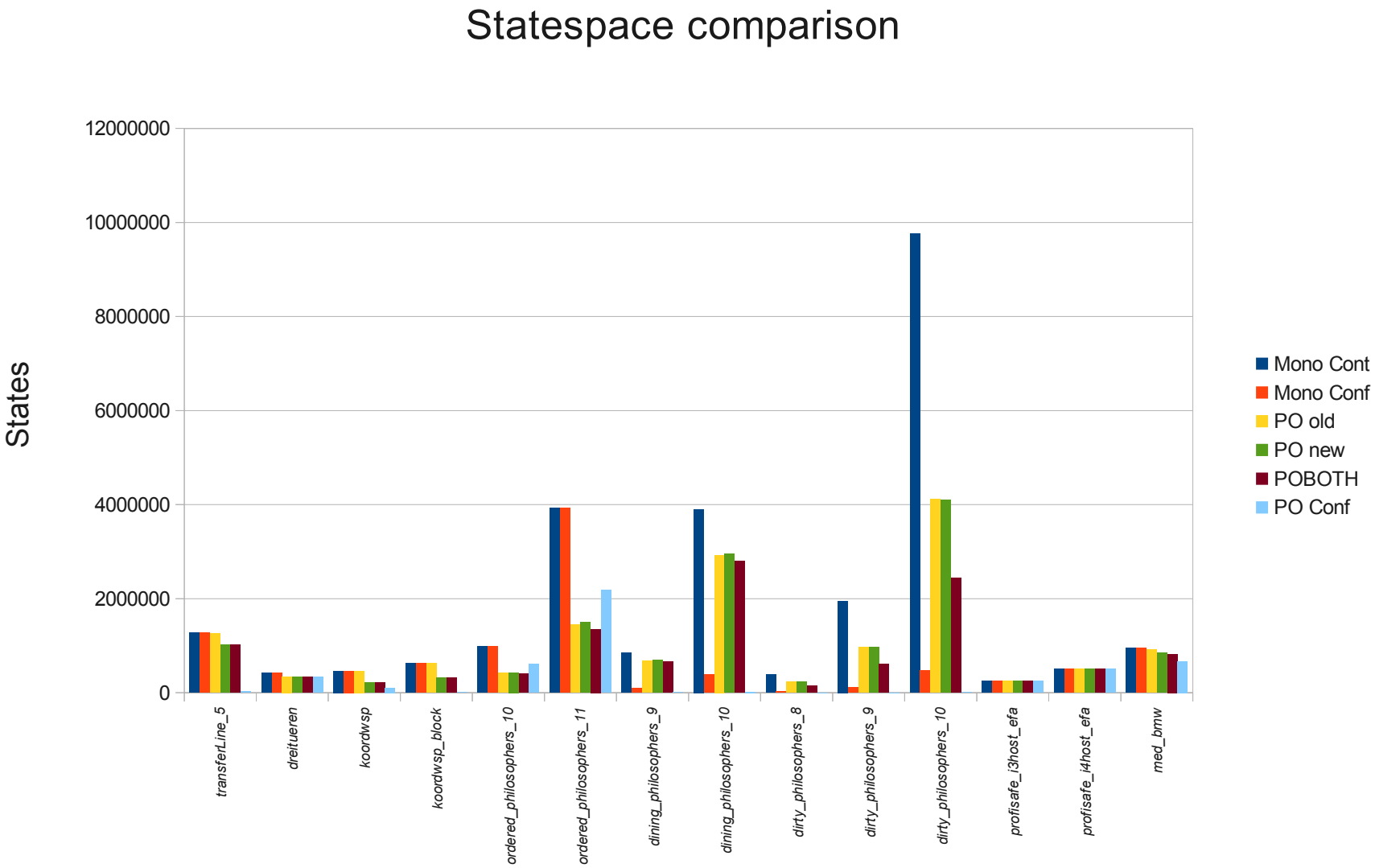

Figure 3.8: Chart showing state-space for models using various tests.

3.4 Conclusions

The partial order reduction implementations discussed in this chapter have yielded some interesting results. Using the results achieved in this chapter we have been able to verify the effectiveness of several optimisations to the partial order reduction process offered in [17]. Some of these optimisations were hypothesised in [17] and some were developed due to considerations that were made over the course of this research. It remains the case that the partial order reduction method is most effective when verifying models that have a large number of independent events and also when verifying models that exhibit a large amount of symmetric structure. We were also able to verify some negative results showing that partial order reduction can not always be effective for verifying very large models.

The main improvement that can be taken from this chapter is in addressing the main drawback that the implementation offered in [17] had, that being the time taken to verify very large models. Only in a few unique cases did the reduction achieved allow for faster verification, and generally the partial order reduction verifier in [17] performed significantly slower than the standard monolithic verifier. With the implementation of Algorithm 6 in place of Algorithm 5 we have drastically improved the run time of the partial order reduction process. Since the majority of the time spent during this process is in calculating ample sets, providing an effective way to minimise the amount of searching required has had a great effect. The advantages of this new ample algorithm over the original one are two fold. Firstly, as is outlined in Section 3.1, the worst case run time complexity is two orders of magnitude lower. This should yield an immediate benefit for verifying models with large numbers of events, as seen when verifying the profisafe models. Secondly the way in which the ample set is constructed allows for a reduced amount of searching. The original implementation would start with an arbitrary event α from the enabled set of events, and construct the dependent enabled and dependent not enabled sets in order to see if an ample set containing α is valid. If it was not valid however, we would continue to

just keep adding events from the enabled set of events to ample until either we had a valid ample set, or the ample set contained all of the enabled events. In this new implementation, once an ample set has been checked and discarded, we start with a fresh ample set and we never consider an event that has already appeared in a previously discarded ample set. This is because of the reflective nature of the dependency relation, if we have chosen α for ample and β depends on α, then we have subsequently discarded that ample set, we should no longer choose β for ample as it will generate the identical ample set as the one that was discarded. While the events may still be chosen arbitrarily from the enabled set of events, these optimisations generally allow for smaller ample sets and much less computational overhead.

Addressing the arbitrary nature by which the events were selected for ample was one of the issues raised in [17]. We have explored two such ways to do that during this research with mixed results. Initially it seemed like this would be a defining factor in generating the smallest possible ample sets. In reality however it seems that the number of viable ample sets, generally speaking, for some state, is not great enough to yield consistently smaller ample sets. This is further highlighted in Table 3.2 where it can be seen that for all models that were verified, it made no difference at all which selection strategy was used, even in the cases where it did seem to reduce the state space from an arbitrary selection of events. This may be due to the way in which the selection strategies were implemented, and there may be more effective selection strategies to attempt, though it seems that models with a consistently relatively high number of potential ample sets may be required in order to effectively test these different strategies.

The introduction of the component condition and an interative version of Tarjan's algorithm to perform the depth first search were born out of another concern highlighted in [17], that of the high number of fully expanded states due to the cycle condition **C3**. These developments certainly had the effect that was intended though the result was not as significant as suspected. We can see from Tables 3.2 and 3.3 that of all models verified using the

component condition **C4** instead, none of them were forced to fully expand any states. This is due to the fact that we recorded inside the state whenever it was the case that an ample set was equal to the enabled set for that state, then when it came time to close a component, if t was determined that any of the states in that component had already been fully expanded incidentally, we did not need to do anything in order to satisfy **C4′**. As has been mentioned this did result in eliminating the need to fully expand states, however the results show that except in a few cases, this has not had a significant impact on the state space. The strongly connected component approach however did allow for a simple implementation of a nonblocking verifier with partial order reduction. This proved to be an effective verification tool as in all blocking models, the nonblocking component check given in Definition 3.3 was able to be determined very quickly after exploring on average about 95% fewer states than the monolithic nonblocking verifier. The partial order reduction process as it is implemented in this chapter appears to be better suited to nonblocking verification than it is for controllability verification. This is due the the reason explained in Section 3.3, where the plantification process increases the number of dependencies in the model which leads to fewer reduced ample sets. Worth particular mention is the transferLine series of tests with the nonblocking partial order reduction verifier. This was so successful that is could be worth examining these models specifically to try and determine the best ways to go about creating ample sets.

There are several avenues in which further work could be done in improving the partial order reduction verifiers discussed in this chapter. While the ample algorithm seems to be quite efficient, there still remains work to be done in finding an optimal strategy for ordering the events. Once such strategy might be to consider altering the transition relation with a new abstraction by grouping events, and considering the independence of the groups. Since there are a vast number of ways in which events could be grouped, a routine may be developed to examine the existing events and determine the best, or at least a good grouping of the events that might yield a higher number of independencies. Another improvement targetted at improving the

controllabiity verifier might be to consider a different abstraction that does not introduce additional dependencies into the model. It may be the case that these dependencies are required in order to safely verify for controllability, but it is worth exploring this as an option. It may also be worth spending time looking at the particular models that exhibited both good and bad behaviours when checked by the partial order reduction verifiers. The specific models in this case would be transferLine and ordered_Philosophers when checked for nonblocking. As has been mentioned determining precisely what allows the greatest reduction, or inhibits it, could prove useful when making further considerations for optimising the partial order reduction process.

Chapter 4

Partial Order Reduction in Compositional Verification

This chapter will introduce and discuss the work done in developing a model verifier with elements of both partial order reduction and compositional verification. This will take each of those concepts and provide an original abstraction that can be used for nonblocking compositional verification. This chapter is arranged similarly to Chapter 3 with sections for algorithms, proofs, results and conclusions. Section 4.1 will detail the various algorithms that were created in order to develop the model verifier. The algorithms offered in Section 4.1 are based on those described in [9] for a compositional model verifier for nonblocking, and it will be made clear where the original work has been introduced. Section 4.2 will provide an original proof for the correctness of the abstraction used as the basis for the model verifier described in this chapter. Section 4.3 will summarise the research done, reflect on what was achieved and discuss the possibilities for further research in this area.

Before beginning a discussion on the algorithms developed for this chapter it will be valuable to first introduce the concepts upon which they are based and the motivations behind them.

As we have established in Section 2.6, compositional verification operates by applying different rules repeatedly to the component automata that are

being synchronised. This is often done by identifying states that match certain criteria and merging them into a single state. What this is in effect doing is creating an abstraction of the automaton such that the abstraction and the original automaton are equivalent in some way with respect to the property that is being verified. One such rule that was introduced in Section 2.6 was the silent continuation rule, where the equivalence relation used is conflict equivalence. It was mentioned that the silent continuation rule was used as a basis for work described in this chapter, we will now now introduce this new abstraction and explain how it is derived from the silent continuation rule.

We use the silent continuation rule which allows us to merge states that are incoming equivalent and able to reach one or more stable states using only silent transitions, with the idea of independence as used in partial order reduction. This new abstraction differs uniquely to the other compositional verification rule in that it is applied as two component automata are being synchronised. We first observe that when two component automata $A_1 = \langle \Sigma_1, S_1, \rightarrow_1, S_1^\circ \rangle$ and $A_2 = \langle \Sigma_2, S_2, \rightarrow_2, S_2^\circ \rangle$ are synchronised, the τ event used in transitions of A_1 is guaranteed to be independent of the τ event used in transitions of A_2. This is because the τ event is used exclusively in transitions where events local to that automaton would be used. For the sake of explanation we will refer to transitions involving τ from A_1 as using event τ_1 and similarly for transitions involving τ from A_2 we will use event τ_2, even though in practice there is only one τ event. To show how this independence is guaranteed, consider an arbitrary state (x_1, x_2) from the synchronous product of $A_1||A_2$, where we have $x_1, y_1 \in S_1$, $x_2, y_2 \in S_2$, $x_1 \xrightarrow{\tau}_1 y_1$. and $x_2 \xrightarrow{\tau}_2 y_2$. We then have an arbitrary state where both τ_1 and τ_2 are enabled. We will then have transitions $(x_1, x_2) \xrightarrow{\tau_1} (y_1, x_2)$, $(x_1, x_2) \xrightarrow{\tau_2} (x_1, y_2)$, $(y_1, x_2) \xrightarrow{\tau_2} (y_1, y_2)$, and $(x_1, y_2) \xrightarrow{\tau_1} (y_1, y_2)$. This is the independence diamond that was introduced in Section 2.5. We propose that in each such state where τ events from the different automata are enabled and this independence diamond exists, we can exclude the intermediate states (x_1, y_2) and (y_1, x_2), and instead create the transition $(x_1, x_2) \xrightarrow{\tau} (y_1, y_2)$. In order to do this

while preserving conflict equivalence we must preserve the transitions to the successor states of the intermediate states that are being excluded. This is where we borrow from the silence continuation rule. We identify each of the stable states reachable from (x_1, x_2) and consider the paths to those states. We then take all of the states on those paths and take their non τ transitions, then add those transitions to (x_1, x_2). That way we preserve the reachability of states that could potentially otherwise only be reachable from the excluded states. An example of this process is given in Figures 4.1, 4.2, and 4.3. Figure 4.1 shows two component automata A_1 and A_2. Each have one local event and synchronise on event α. Figure 4.2 shows the synchronous conposition $A_1||A_2$. In this we can see the independence diamond formed by the local events τ_1 and τ_2. We also have successor states using event α from each of the intermediate states (x_1, y_2) and (y_1, x_2) of the diamond. Finally in Figure 4.3 we have the abstraction of $A_1||A_2$ using the process described above. Here we see that the intermediate states (x_1, y_2) and (y_1, x_2) have been excluded, and the transition $(x_1, x_2) \xrightarrow{\tau} (y_1, y_2)$ added. Also we can see that we now have three transitions on event α from (x_1, x_2). These transitions have been taken from the intermediate states. Consider the intermediate state (x_1, y_2). The only stable state reachable from this state is the state (y_1, z_2). Following the process that has been described, we then examine each state on the path p with $(x_1, y_2) \xrightarrow{p} (y_1, z_2)$. Every non τ transition is then added to state (x_1, x_2). The only such transition is $(x_1, y_2) \xrightarrow{\alpha} (x_1, z_2)$, which leads to the creation of the transition $(x_1, x_2) \xrightarrow{\alpha} (x_1, z_2)$ in the abstraction. The same process happens to the opposite intermediate state (y_1, x_2) leading to the creation of the transition $(x_1, x_2) \xrightarrow{\alpha} (z_1, x_2)$. Note that the τ_1 and τ_2 events have been replaced by the τ event in the abstraction, as once the abstraction is completed and there are no longer any independencies to identify, there is no longer any need to distinguish between them. A formal definition of this abstraction is given in the following definition.

Definition 4.1. Let $A_1 = \langle \Sigma_1, S_1, \rightarrow_1, S_1^\circ \rangle$ and $A_2 = \langle \Sigma_2, S_2, \rightarrow_2, S_2^\circ \rangle$ be two automata. Let $G = A_1||A_2 = \langle \Sigma, S, \rightarrow_G, S^\circ \rangle$ and $H = \langle \Sigma, S, \rightarrow_H, S^\circ \rangle$ with

$$\rightarrow_H = \{((x_1, x_2), \sigma, (y_1, y_2)) \mid x_1 \xrightarrow{\sigma}_1 y_1, x_2 \xrightarrow{\sigma}_2 y_2, \sigma \in \Sigma_1 \cap \Sigma_2 \vee \sigma = \tau\}$$
$$\cup \{((x_1, x_2), \sigma, (y_1, x_2)) \mid x_1 \xrightarrow{\sigma}_1 y_1, \sigma \in \Sigma_1 \setminus \Sigma_2 \vee (\sigma = \tau \wedge x_2 \not\xrightarrow{\tau}_2)\}$$

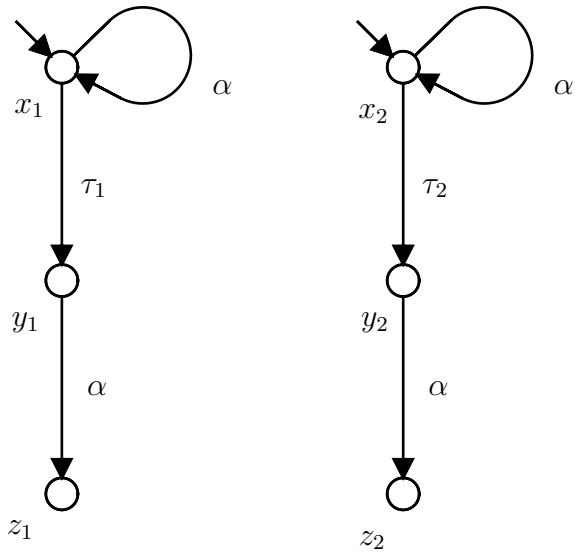

Figure 4.1: Component automata A_1 and A_2

$$\cup\ \{((x_1,x_2),\sigma,(x_1,y_2)) \mid x_2 \xrightarrow{\sigma}_2 y_2, \sigma \in \Sigma_2 \setminus \Sigma_1 \vee (\sigma = \tau \wedge x_1 \not\xrightarrow{\tau}_1)\}$$
$$\cup\ \{((x_1,x_2),\sigma,(y_1,z_2)) \mid x_1 \xrightarrow{\tau}_1, x_1 \xrightarrow{\sigma}_1 y_1, \sigma \neq \tau, x_2 \xrightarrow{\tau^+}_2 y_2 \xrightarrow{\sigma}_2 z_2\}$$
$$\cup\ \{((x_1,x_2),\sigma,(y_1,z_2)) \mid x_2 \xrightarrow{\tau}_2, x_2 \xrightarrow{\sigma}_2 y_2, \sigma \neq \tau, x_1 \xrightarrow{\tau^+}_1 y_1 \xrightarrow{\sigma}_1 z_1\}$$

Then H is the abstraction obtained from G on application of the *silent continuaton with independence* rule applied to all appropriate states.

One issue does remain however in borrowing from the silent continuation rule in order to preserve the non τ successors of the states on the paths to the stable states. That issue is that the silent continuation rule requires incoming equivalence between those states with what would be state (x_1, x_2) in the example given. As can be seen from the definition this is not a requirement of this abstraction. Figures 4.4, 4.5 and 4.6 demonstrate the abstraction process when incoming equivalence is not present in the process. Figure 4.4 shows the component automata A_1' and A_2' which are A_1 and A_2 extended to include some extra behaviour that will generate incoming transitions for the intermediate states in the independence diamond of the synchronous composition. These automata still only have one local τ event each but they

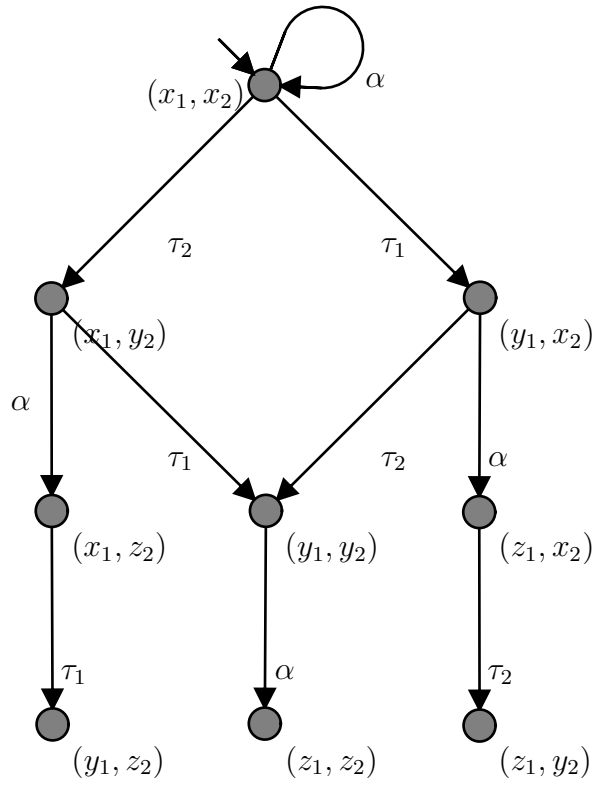

Figure 4.2: Regular synchronous composition $A_1||A_2$

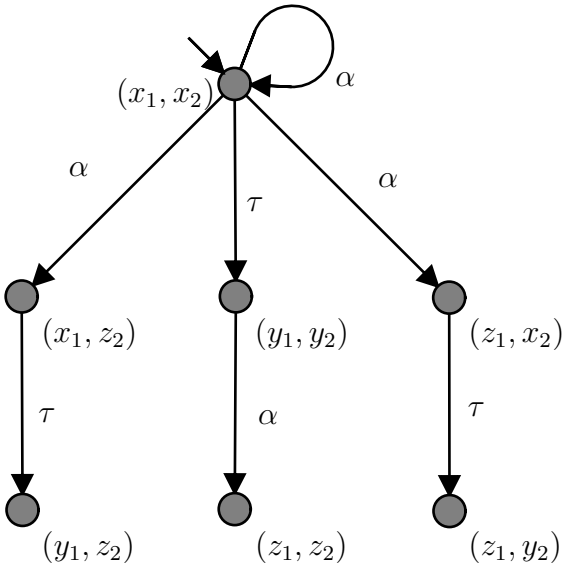

Figure 4.3: Abstraction of $A_1||A_2$ under new rule

now synchronise on four distinct events α, β, γ and σ. Figure 4.5 shows the synchronous composition $A_1'||A_2'$. It can be seen that the independence diamond formed by τ_1 and τ_2 is still present, though now there are two new successor states from the initial state (x_1, x_2) given by $(x_1, x_2) \xrightarrow{\beta} (u_1, u_2)$ and $(x_1, x_2) \xrightarrow{\alpha} (v_1, v_2)$. These new states are the states from which the incoming transitions to the intermediate states (y_1, x_2) and (x_1, y_2) originate. Each of these states has two outgoing transitions, one to each intermediate state. The result of this is each intermediate state having two non τ incoming transitions, a σ transition and a γ transition. These transitions mean that the intermediate states (y_1, x_2) and (x_1, y_2) are not incoming equivalent to the initial state (x_1, x_2). Figure 4.6 shows the abstraction of $A_1'||A_2'$ with the silent continuation with independence rule. Immediately we see that the state-space has remained unchanged, though the transitions have been altered. To understand what has happened here it will be helpful to step through the process of constructing the automaton given in Figure 4.6. Starting at the initial state (x_1, x_2) we calculate all the successors as determined by the transition relation given in Definition 4.1. This yields the transitions

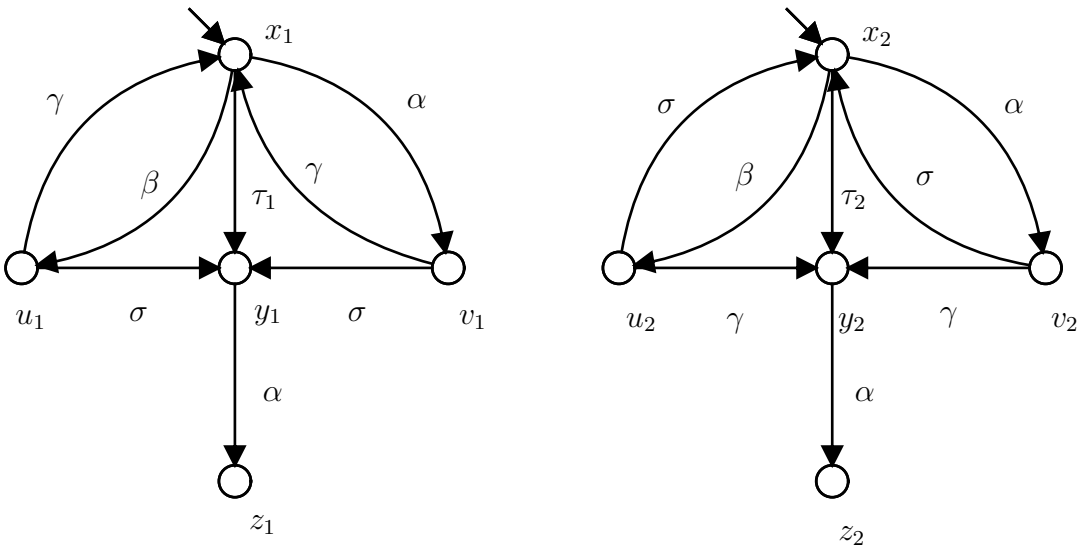

Figure 4.4: Component automata A_1' and A_2'

$(x_1, x_2) \xrightarrow{\tau} (y_1, y_2)$, $(x_1, x_2) \xrightarrow{\beta} (u_1, u_2)$, $(x_1, x_2) \xrightarrow{\alpha} (v_1, v_2)$, $(x_1, x_2) \xrightarrow{\alpha} (z_1, v_2)$, and $(x_1, x_2) \xrightarrow{\alpha} (v_1, z_2)$. At this point it is clear that as in the previous case, the intermediate states have been excluded. Next we can consider any of the successor states that were computed and explore those. What we find here is that since states (u_1, u_2) and (v_1, v_2) have transitions to states (x_1, y_2) and (y_1, x_2), those states that were previously excluded are then added to the state-space once (u_1, u_2) and (v_1, v_2) are expanded. What this means is the states that would otherwise not be eligible for silent continuation due to not having incoming equivalence, are able to be considered for silent continuation with independence. This is because of the fact that when those transitions that violate the incomng equivalence relationship are explored, the states that were previously excluded by the silent continuation with independence rule are then added to the state-space. A proof showing the correctness of this abstraction for conflict equivalence is given in Section 4.2.

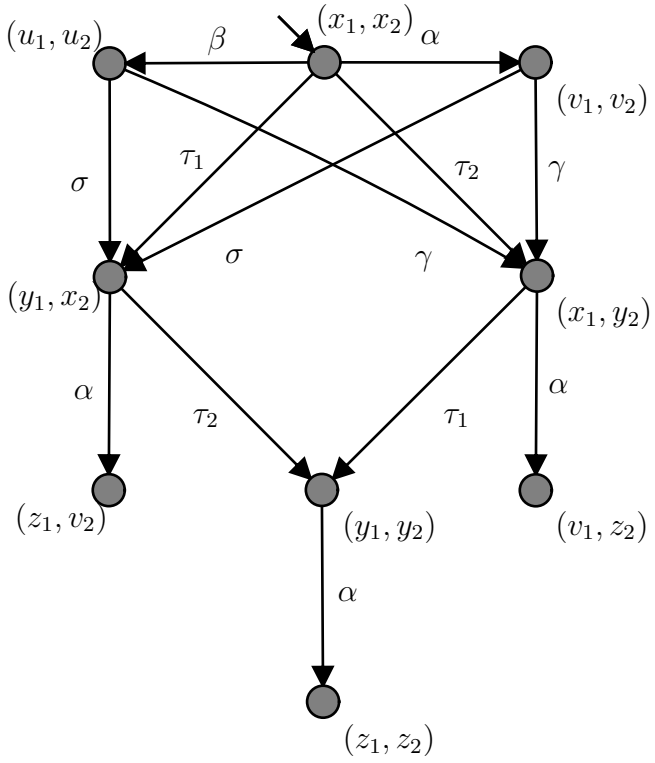

Figure 4.5: Regular synchronous composition $A_1' || A_2'$

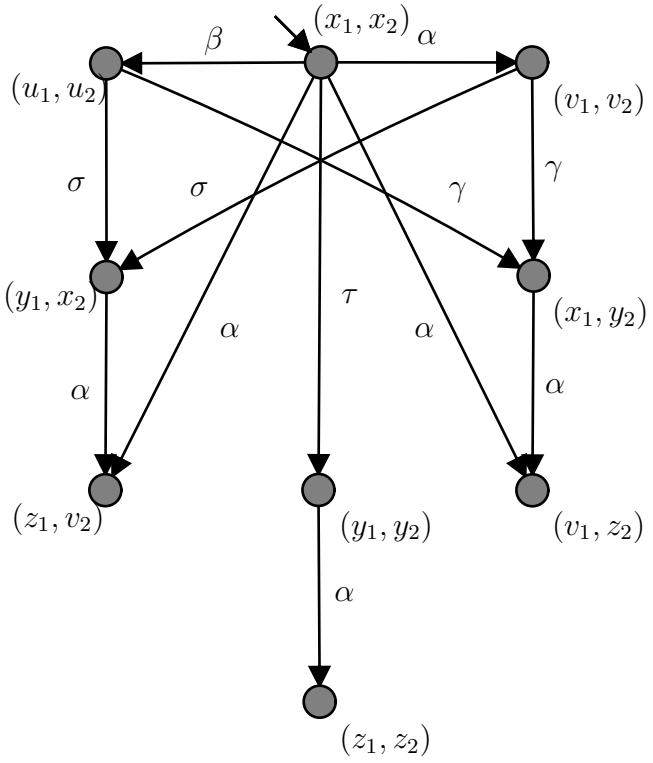

Figure 4.6: Abstraction of $A_1' || A_2'$ under new rule

4.1 Algorithms

This section introduces the algorithms developed in order to construct synchronous compositions for nonblocking verification using the silent continuation with independence rule. This work builds upon the general synchronous product builder given in [9] and adjusts the way in which successor states of states that the transition relation given in Definition 4.1 identifies. This implementation is used for nonblocking verification, so as with the implementation in Chapter 3, the algorithms used are such that the resulting automaton that is constructed preserves the property of nonblocking that the regular synchronous composition would exhibit.

4.1.1 Tau closure algorithm

One of the key parts to applying this rule is finding all of the τ successors of the states in the model. By τ successors we mean all of the states that can be reached from some state using only τ transitions. The reason this is important is that for each of the states in the synchronous composition that enable more than one local τ event, we need to find all of the τ successor states, and then for each of those successor states the non τ transitions must be added to the state that is being explored. The way in which this has been handled in this implementation is by modifying the transition relation in the original model to include the τ closure for each state. That is to say that for every state x of every automaton in the model, every state y reachable using only τ transitions from state x is then added as the target state of a new τ transition from state x. Figures 4.7 and 4.8 give an example of an automaton before and after performing the τ closure on each of the states. It can be seen in Figure 4.7 that there exist paths p, q, and r consisting of only τ transitions where $x \xrightarrow{p} y_1$, $x \xrightarrow{q} y_2$ and $x \xrightarrow{r} y_3$. Figure 4.8 then shows the τ closure of x adding transitions $x \xrightarrow{\tau} y_1$, $x \xrightarrow{\tau} y_2$, and $x \xrightarrow{\tau} y_3$. Since the τ closure is performed on all states we also have the $y_1 \xrightarrow{\tau} y_3$ transition added.

Algorithm 10 gives the algorithm used to calculate the τ closure and

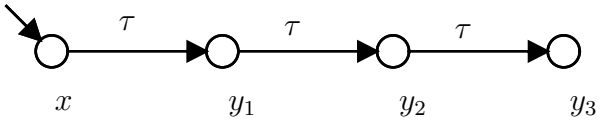

Figure 4.7: Simple automaton prior to τ closure

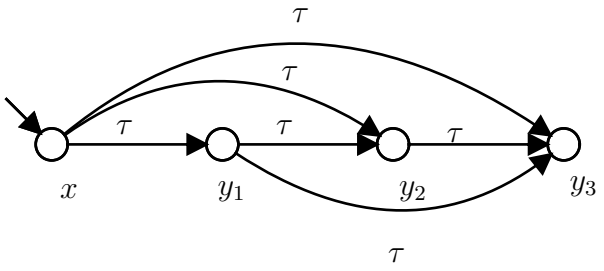

Figure 4.8: Automaton after τ closure

update the transition relation. This algorithm explores the state space of each component automata performing a depth first search using only τ transitions. To accomplish this we first iterate over all automata in the model and then over all states in those automata. The search begins as each state s is considered. We initialise two variables; *stack* which will serve as a state stack to keep track of the states still to be explored in the depth first traversal, and *visited* which will serve as the set of states that have been visited in the depth first traversal. These are both initialised with state s included in them. Then comes the main loop in the depth first traversal, where we loop until *stack* is empty. At each iteration of this loop we pop a state *current* from the top of *stack* and consider all of the τ successors of *current*. Since the hiding process replaces all local events in an automata with τ, this will often lead to non determinism, so there may be several τ transitions from state *current*. For each τ successor t of state *current* we then check to see whether or not $t \in visited$. This makes sure that we do not return to explore states that have already been explored. Every successor $t \notin visited$ is then added to both *stack* and *visited*. At the end of this process *visited* will

contain all states that are reachable from state s using only τ transitions, and we can use those states to perform the τ closure of s.

First state s is removed from *visited*, as we do not want to add the transition $s \xrightarrow{\tau} s$ to the transition relation. Once this is done we then iterated over all states in *visited*. For each such state t we check whether or not there is not already a transition $s \xrightarrow{\tau} t$. If there is no such transition, we add it to the transition relation, otherwise we move on to the next t. Once this has been done, the τ closure of state s is complete and the transition relation has had all of the appropriate transitions added. We then move on to the next state s and reeat until all states in all automata have had their τ closure performed. This will allow for a simpler calculation of τ successor states when constructing the synchronous composition.

4.1.2 Permute non tau successors

As it has been suggested, by performing the hiding process we often introduce non determinism into the component automata of the synchronous product that we are creating. As such, this means that as we explore states we are no longer guaranteed that there will be a single successor state for a transition on an event. Not only may there be several successor states for an event in a component automaton, when we consider a global state of the synchronous product comprised of several local states of the component automata, the possible successor states for that event for the global state are the combination of all the different transitions on that event in the component automata. Let $(x_0, x_1, \ldots, x_n)$ be a global state of the synchronous product $A_1||A_2||\ldots||A_n$ and $|x \xrightarrow{\alpha} |$ be the number of outgoing α transitions from state x, then the number of successor states on event α from state $(x_0, x_1, \ldots, x_n)$ is given by $\prod_{i=0}^{n} |x_i \xrightarrow{\alpha} |$. An example of this is given in Figures 4.9 and 4.10. Figure 4.9 gives two simple nondeterministic component automata A_1 and A_2 and Figure 4.10 gives the expansion of global state (w_1, w_2) to find all successor states. It can be seen from Figure 4.10 that there are four successor states, two for each event. If we consider what happens when

Algorithm 10 Tau closure algorithm

```
1: function TauClosure
2:     for all Automata A = ⟨Σ, S, S°, →, Q⟩ do
3:         for all s ∈ S do
4:             stack = {s}
5:             visited = {s}
6:             while ¬stack.isEmpty() do
7:                 current = stack.pop()
8:                 for all t where current --τ--> t do
9:                     if t ∉ visited then
10:                        stack.push(t)
11:                        visited = visited ∪ {t}
12:                    end if
13:                end for
14:            end while
15:            visited = visited \ {s}
16:            for all t ∈ visited do
17:                if s --τ--/--> t then
18:                    → = → ∪ {(s, τ, t)}
19:                end if
20:            end for
21:        end for
22:    end for
23: end function
```

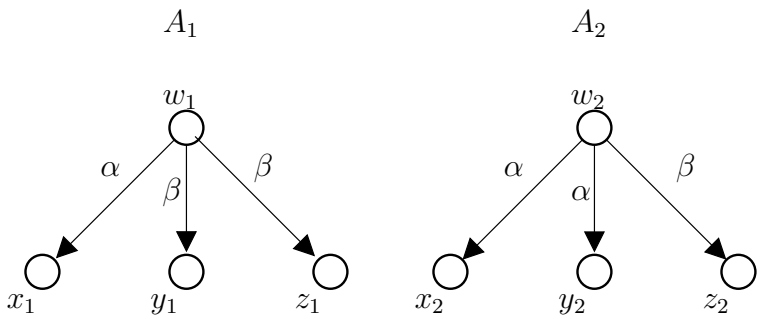

Figure 4.9: Nondeterministic automata A_1 and A_2

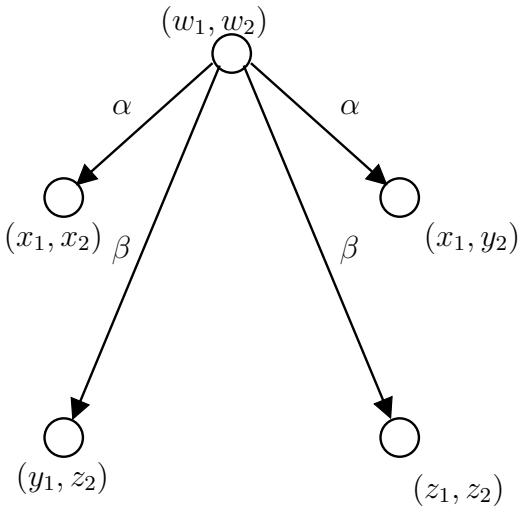

Figure 4.10: Expansion of state (w_1, w_2) in $A_1||A_2$

determining the successor states for event α, we can observe from 4.9 that A_1 has only one such transition $w_1 \xrightarrow{\alpha} x_1$, whereas A_2 has two transitions $w_2 \xrightarrow{\alpha} x_2$ and $w_2 \xrightarrow{\alpha} y_2$. This gives two possible state combinations for the global successor state. A_1 will always do its only α transition to state x_1, but A_2 can perform either of its two α transitions. This leads to global successor states (x_1, x_2) and (x_1, y_2) for event α, as seen in Figure 4.10. The successor states (y_1, z_2) and (z_1, z_2) for event β can be found in a similar way.

Clearly then an algorithm is required to compute all of these successor states accounting for the nondeterminism. Algorithm 11 gives the psuedo

code for the algorithm that generates all of the permutations of the successor states for some state s in a nondeterministic synchronous composition and creates the transitions to those states. This algorithm is taken from [9]. Generally speaking the way in which this algorithm works is by keeping track of a global target state t which will be used to created each successor state from s. First the enabled set of events for state s is computed and we set $t = s$ before computing the successors for each event. Then $PermuteNormal$ is recursively called for each enabled event α, which at each step changes decreases the variable a by 1, and a single local state in t indexed by a, so by the time a reaches zero every local state has been updated to reflect a transition on event α in the component automata. We will now offer a more detailed description of the algorithm.

$ExpandNormal$ just serves to calculate the enabled set of events for state s and compute the successors for each such event. Before $PermuteNormal$ is called global state variable t is initialised and given the same value as s, this way if some automaton does not contain the enabled event in its alphabet, we can preserve the state for that automaton. We call $PermuteNormal$ for each enabled event alpha passing in $n + 1$, s and $alpha$. The first argument is one more than the number of component automata, as the first thing to happen in $PermuteNormal$ is to decrement a which will initially receive the value $n+1$, so once that happens the very first consideration for target states will be for state x_n. Now lets examine the $PermuteNormal$ function. Of the four arguments, a is the one of most interest. There are two states s and $source$ passed in as one is used for creating the transition and the other is used for computing local successor states; it will be clearer as to why we need both of these when we discuss Algorithm 12. The variable a in this function represents the index of the local automaton that the current function frame is to consider. The initial line checks whether or not $a = 0$. We will come back to this point once the rest of the routine has been explained. In the case that $a \neq 0$ we proceed to decrement a by 1. This means that on each subsequent call to $PermuteNormal$ as long as $a > 0$, we update a to represent the index of a different component automaton. Next we examine the local state x_a of

component automaton M_a. We have already established that α is enabled in state s, though it is still potentially the case that $x_a \overset{\alpha}{\not\rightarrow}$. For this to be the case we must have $\alpha \notin \Sigma_a$, and as such a transition on event α does not affect state x_a. We then update t accordingly, setting $x'_a = x_a$, before making a recursive call to $PermuteNormal$. If however it is the case that $x_a \overset{\alpha}{\rightarrow}$ then we must consider all of the successor states of for x_a using event α. Again there may be several of these successor states due to the nondeterminism of the component automata after hiding. We then consider each local state y where $x_a \overset{\alpha}{\rightarrow} y$, and update the target state t setting $x_a = y$. We then make a recursive call to $PermuteNormal$ before doing the same for the next y. What is happening here is that since every distinct local successor state generates at least one distinct global successor state, we record that local transition in the global target state t and then move on to consider the changes in all of the other local states. Since we start this process at state x_n and proceed until x_0, by the time the first recursive call resolves when considering the successors of state x_n, we will have computed all possible global successor states for the the first local successor of x_n on event α. The reason for this is more easily explained considering the process from the end of the recursion and working backwards. When $a = 0$ this means that all local states $x_n \ldots x_0$ have been updated in t, so the current state of t represents a valid successor state in the synchronous composition, so the transition $source \overset{\alpha}{\rightarrow} t$ can be created. When $a = 1$, we first decrement a to 0 and proceed to calculate the successor states for local state x_0. Each of the states computed make a call to $PermuteNormal$ where $a = 0$, creating a global successor state for every such call. Consider the state of the algorithm from where the $a = 1$ recursive call was made however. It was called after calculating just one of the local successor states for x_1. This means that for every successor of x_1, all successors of x_0 are computed. Going one step further up the recursion gives us that for every successor of x_2, all successors of x_1 are computed, which in turn each compute the successors of x_0. This propogates all the way up to the first call where $a = n$ meaning that by the time all successors of a_n have been computed, we have created transitions to every global successor state t where $source \overset{\alpha}{\rightarrow} t$.

Algorithm 11 Permute non tau successor states

```
1: function ExpandNormal(s = (x_0, ..., x_n))
2:     for all α ∈ Σ where s -α-> do
3:         t = (x'_0, ..., x'_n) = s
4:         PermuteNormal(n + 1, s, s, α)
5:     end for
6: end function
7: function PermuteNormal(a, s = (x_0, ..., x_n), source, α)
8:     if a = 0 then
9:         createTransition(source, α, t)
10:    else
11:        a = a − 1
12:        if x_a -α-/-> then
13:            t = (x'_0, ..., x'_a = x_a, ..., x'_n)
14:            PermuteNormal(a, s, source, α)
15:        else
16:            for all y ∈ S_a where x_a -α-> y do
17:                t = (x'_0, ..., x'_a = y, ..., x'_n)
18:                PermuteNormal(a, s, source, α)
19:            end for
20:        end if
21:    end if
22: end function
```

4.1.3 Permute tau successors

The algorithms described in this section is similar to Algorithm 11 but a little more involved. The purpose here is to manufacture the new transitions given in Definition 4.1. That is to say that when we encounter states that have two or more τ events enabled, we must make sure to exclude the transitions to the intermediate states, copy all outgoing transitions from the intermediate states to the source state, and create a new τ transition from the source state to the state reached once all of the enabled τ transitions have been taken. To find these intermediate states we must permute the different combinations of taking the enabled τ transitions. Figure 4.11 shows an example of the expansion of a state with three different τ events enabled. In this example the intermediate states are any of the states containing a combination of x_i and y_i local states. We need to calculate these states, find their outgoing transitions, and add them to state (x_0, x_1, x_2). We must also create the transition $(x_0, x_1, x_2) \xrightarrow{\tau} (y_0, y_1, y_2)$. It can be seen from Figure 4.11 that each of these states are reached by performing τ_1, τ_2 and τ_3 in different orders, suggesting that we must permute all the combinations of the enabled τ events in order to calculate all of the intermediate states. Notice however that this example is deterministic. As was the case before, since we are dealing with nondeterministic component automata we must also allow for the fact that there may be several τ transitions in the same local state and calculate each of the intermediate states reached using these transitions as well. The number of intermediate states to be found assuming determinism is $n!$ where n is the number of different enabled τ events.

Algorithm 12 gives the psuedo code for performing the process described above. This is similar to the process described for Algorithm 11, however where Algorithm 11 permutes the successor states for a single event at a time, Algorithm 12 instead permutes the different enabled τ events to find the successor states. The algorithm begins in *ExpandTau* where a collection of events *enabledTau* is initialised to store each of the different enabled τ events of state s. Once the enabled τ events have been determined we check

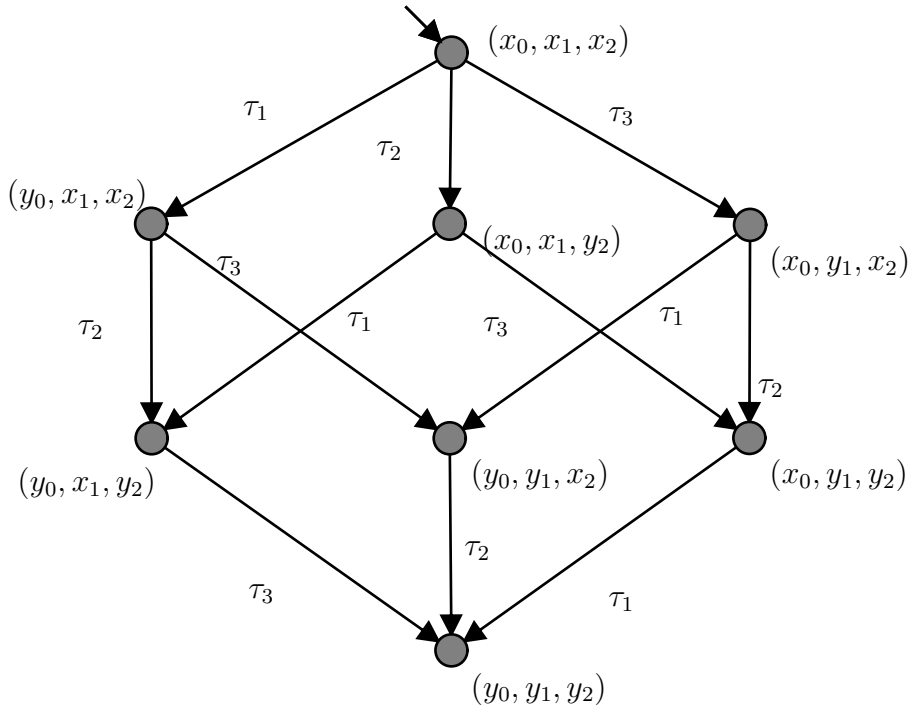

Figure 4.11: Result of expanding state with three different τ events enabled

to see whether there are either two or more enabled τ events, or a single enabled τ event. If there is just a single τ event enabled then we can treat this as a normal transition and simply make a call to $PermuteNormal$, which will generate the appropriate successor states for s. If there are two or more enabled τ events then we make a call to $PermuteTau$ passing in the number of enabled τ events, the collection $enabledTau$, two copies of the state s being expanded, and the boolean value $true$ for the last two arguments. It will be made clear exactly what these variables are used for in the description of $PermuteTau$.

As was mentioned above, $PermuteTau$ differs from $PermuteNormal$ in that it is permuting the order in which the different τ transitions are performed. As such, the integer value t that serves as the end point for the recursion is no longer the index of a component automaton, but the number of enabled τ events. Now as t decreases we are using it to select the next τ event from $enabledTau$, although this does also have the effect of selecting a component automaton however as a τ event will only exist in the alphabet of a single automaton. Also the role of state s changes in $PermuteTau$. Since we are no longer just calculating direct successors of $source$ and we may now need to calculate intermediate states that are reached after several transitions from $source$, we need to save the changes we make to s and use the updated state s in order to calculate further states. New to $PermuteTau$ also are the boolean variables $first$ and $last$. These variables are used to determine whether or not a state that has been calculated is one of the intermediate states, the source state, or the final state once all τ transitions have been performed.

The initial line checks to see if $t = 0$. If this is true then this signifies the end of recursion at which point state s will be a potential intermediate state. The further check of $\neg first \wedge \neg last$ determines if s is an intermediate state or not, which will be explained shortly. If $t \neq 0$ then we first decrement t by 1 and make a recursive call to $PermuteTau$. We do this before making any changes to s because we must consider the intermediate states where some

of the τ transitions have not been taken. To use the example in Figure 4.11, state (y_0, x_1, x_2) is one of the intermediate states, but only the τ_1 transition has been taken. This means that for the two values of t representing events τ_2 and τ_3, we need to make a call to $PermuteTau$ before any changes are made to s, which is what happens on line 27. Note that this will include the permutation of none of the τ transitions being taken as one of the potential solutions. Obviously this is not one of the intermediate states as nothing has changed, so this solution must be skipped. This is handled by the boolean variable $first$. Initially $first$ has the value $true$ passed in from $ExpandTau$. It can be seen from the two recursive calls to $PermuteTau$ that line 27 preserves the value of $first$ whereas line 33 passes in $false$ in place of $first$. This means that the only way $first = true$ by the time $t = 0$ is if none of the recursive calls from line 33 have been made. Of course if this is the case then we have the case that was just described where none of the τ transitions have been taken, so we can conclude that if $first = true$ when $t = 0$ then $s = source$, and as such should be skipped. After the recursive call on line 27 we proceed to calculate successor states for the τ event indexed by t in $enabledTau$. We get the event and store it in event variable α and then we determine the component automaton index a of the automaton whose alphabet contains α. Since τ events are local events it is guaranteed that there is only one such automaton. We then store a backup of x_a in the variable $backup$ so that we may restore the state s to its original state once all of the successors have been computed. We then consider each local state y where $x_a \xrightarrow{\alpha} y$, and update state s setting $x_a = y$. We then make a recursive call to $PermuteNormal$ before doing the same for the next y. The way this works is the same as in $PermuteNormal$ except for the boolean variables $first$ and $last$. It has been explained how the $first$ variable is used to determine if the state that has been calculated is the source state, but it remains to be shown how the $last$ variable is used. Similarly to how $first$ was handled it can be seen that line 33 preserves the value of $last$ whereas line 27 passes in $false$ in place of $last$. This means that the only way $last = true$ by the time $t = 0$ is if none of the recursive calls from line 27 have been made. The effect of this is that if $last = true$ and $t = 0$, then every τ event in $enabledTau$ has been used

and its successor state included in state s, meaning that we have identified the final state s once all τ transitions have been performed. This state is a special case as we do not copy its outgoing transitions back to the start state, rather we create the transition directly to it from source with event τ. Since we know that $last = true$ and $first = true$ do not yield intermediate states, we then have that if $\neg first \wedge \neg last$ when $t = 0$ then we have identified an intermediate state. At this point it remains to copy all non τ outgoing transitions from this state s back to *source*. To do this we make a call to $AddTauSuccessors(s, source)$. $AddTauSuccessors$ works very similarly to $ExpandNormal$ but with one critical difference. Each enabled event of state s is passed in to $PermuteNormal$ again with a fresh target state t initialised as $t = s$, however now s and *source* are not the same state. The effect of this is that once a successor state t is found in $PermuteNormal$, now the transition that is added is $(source, \alpha, t)$ where *source* is the original state s that was passed in to $ExpandTau$. Once this has happened for every enabled non τ event, we will have successfully created transitions to every non τ successor of the intermediate state found in $PermuteTau$ from state *source*.

4.2 Proof of Correctness

This section provides a proof for the correctness of the abstraction achieved by application of the silent continuation with independence rule described in Definition 4.1. This is an original proof which consists of two lemmas, Lemma 2 and Lemma 3, and Theorem 7.

Lemma 2 proves that if a state y can be reached from state x in the abstraction H using some path p, then the same state y can also be reached from state x in the original automaton G using the same path, but with an arbitrary number of τ events shuffled in. Lemma 3 proves that if a state x can be reached in G from state w using path t, then a state y can be reached in H from state w using a path that contains all of the events of t but not necessarily in the same order, where x is able to reach state y in G using

Algorithm 12 Permute tau successors

```
1: function ExpandTau(s = (x_0, ..., x_n))
2:     enabledTau = ∅
3:     for all α ∈ Σ where s -α-> do
4:         if IsTau(α) then
5:             enabledTau.add(α)
6:         end if
7:     end for
8:     if enabledTau.size() > 1 then
9:         PermuteTau(enabledTau.size(), enabledTau, s, s, true, true)
10:    else
11:        if enabledTau.size() = 1 then
12:            PermuteNormal(n + 1, s, s, enabledTau.first())
13:        end if
14:    end if
15: end function
16: function PermuteTau(t, enabledTau, s = (x_0, ..., x_n), source, first, last)
17:    if t = 0 then
18:        if ¬first ∧ ¬last then
19:            AddTauSuccessors(s, source)
20:        else
21:            if last then
22:                createTransition(source, α, s)
23:            end if
24:        end if
25:    else
26:        t = t − 1
27:        PermuteTau(t, enabledTau, s, source, first, false)
28:        α = enabledTau.get(t)
29:        a = GetAutomatonIndex(α)
30:        backup = x_a
31:        for all y ∈ S_a where x_a -α-> y do
32:            s = (x_0, ..., x_a = y, ..., x_n)
33:            PermuteTau(t, enabledTau, s, source, false, last)
34:        end for
35:        s = (x_0, ..., x_a = backup, ..., x_n)
36:    end if
37: end function
38: function AddTauSuccessors(s, source)
39:    for all α ∈ Σ where s -α-> do
40:        if ¬isTau(α) then
41:            t = (x'_0, ..., x'_n) = s
42:            PermuteNormal(n + 1, s, source, α)
43:        end if
44:    end for
45: end function
```

only silent transitions. Theorem 7 then uses Lemmas 2 and 3 to prove the result claimed in Definition 4.1, that $G \simeq_{conf} H$.

4.2.1 Silent continuation with independence abstraction conflict equivalence

For the proofs given in this section let G and H be the synchronous composition automata as described in Definition 4.1. For clarity then we have $G = A_1||A_2 = \langle \Sigma, S, \rightarrow_G, S^\circ \rangle$ where $A_1 = \langle \Sigma_1, S_1, \rightarrow_1, S_1^\circ \rangle$ and $A_2 = \langle \Sigma_2, S_2, \rightarrow_2, S_2^\circ \rangle$, and $H = \langle \Sigma, S, \rightarrow_H, S^\circ \rangle$.

Lemma 2. If $(x, x_T) \xrightarrow{s} (y, y_T)$ in $H||T$, then $(x, x_T) \overset{P(s)}{\Longrightarrow} (y, y_T)$ in $G||T$.

Proof. Let $s = \sigma_1 \dots \sigma_n$ and $(*)(x, x_T) = (x^0, x_T^0) \xrightarrow{\sigma_1} (x^1, x_T^1) \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} (x^n, x_T^n) = (y, y_T)$. Consider an arbitrary transition $(x^i, x_T^i) \xrightarrow{\sigma_{i+1}} (x^{i+1}, x_T^{i+1})$ from s, where $x^i = (x_1, x_2) \in S$, then one of the following cases must be true:

- $x^{i+1} = (y_1, y_2)$ and $x_1 \xrightarrow{\sigma_{i+1}}_1 y_1, x_2 \xrightarrow{\sigma_{i+1}}_2 y_2$
  - $\sigma_{i+1} \in \Sigma_1 \cap \Sigma_2$. The transition described in this case is part of the standard synchronous composition transition relation and thus $(x^i, x_T^i) = (x_1, x_2, x_T^i) \xrightarrow{\sigma_{i+1}} (y_1, y_2, x_T^i) = (x^{i+1}, x_T^{i+1})$ exists in $G||T$.
  - $\sigma_{i+1} = \tau$. It must be the case here that $x_1 \xrightarrow{\tau}_1$ and $x_2 \xrightarrow{\tau}_2$. Since these events commute in the synchronous product, we have $(x_1, x_2) \xrightarrow{\tau}_G (y_1, x_2) \xrightarrow{\tau}_G (y_1, y_2)$, thus $(x^i, x_T^i) = (x_1, x_2, x_T^i) \overset{P(\sigma_{i+1})}{\Longrightarrow} (y_1, y_2, x_T^i) = (x^{i+1}, x_T^{i+1})$ in $G||T$.
- $x^{i+1} = (y_1, x_2) \wedge x_1 \xrightarrow{\sigma_{i+1}}_1 y_1 \wedge ((\sigma_{i+1} \in \Sigma_1 \setminus \Sigma_2) \vee (\sigma = \tau \wedge x_2 \overset{\tau}{\not\rightarrow}_2))$. The transition described in this case is part of the standard synchronous composition transition relation and thus $(x^i, x_T^i) = (x_1, x_2, x_T^i) \xrightarrow{\sigma_{i+1}} (y_1, x_2, x_T^i) = (x^{i+1}, x_T^{i+1})$ in $G||T$.
- $x^{i+1} = (x_1, y_2) \wedge x_2 \xrightarrow{\sigma_{i+1}}_2 y_2 \wedge ((\sigma_{i+1} \in \Sigma_2 \setminus \Sigma_1) \vee (\sigma = \tau \wedge x_1 \overset{\tau}{\not\rightarrow}_1))$. The transition described in this case is part of the standard synchronous

composition transition relation and thus $(x^i, x_T^i) = (x_1, x_2, x_T^i) \xrightarrow{\sigma_{i+1}} (x_1, y_2, x_T^i) = (x^{i+1}, x_T^{i+1})$ in $G||T$.

- $(\sigma_{i+1} \in \Sigma_T \setminus (\Sigma_1 \cup \Sigma_2)) \vee (\sigma_{i+1} = \tau \wedge x_T^i \neq x_T^{i+1})$. In this case we have $x^i = x^{i+1}$ as the transition is a transition from the test, thus the transition will also be available in $G||T$.

- $x_1 \xrightarrow{\tau}_1, x_2 \xrightarrow{\tau}_2, \sigma_{i+1} \neq \tau$

  - $x_1 \xrightarrow{\sigma_{i+1}}_1 y_1, x_2 \xrightarrow{\tau^+}_2 y_2 \xrightarrow{\sigma_{i+1}}_2 z_2, x^{i+1} = (y_1, z_2)$. In this case we are observing the transition $(x_1, x_2) \xrightarrow{\sigma_{i+1}}_H (y_1, z_2)$ which does not exist in G. By including τ transitions however we have $(x_1, x_2) \xrightarrow{\tau^+}_G (x_1, y_2) \xrightarrow{\sigma_{i+1}}_G (y_1, z_2)$, thus $(x^i, x_T^i) = (x_1, x_2, x_T^i) \overset{P(\sigma_{i+1})}{\Longrightarrow} (y_1, z_2, x_T^i) = (x^{i+1}, x_T^{i+1})$ in $G||T$.
  - $x_2 \xrightarrow{\sigma_{i+1}}_2 y_2, x_1 \xrightarrow{\tau^+}_1 y_1 \xrightarrow{\sigma_{i+1}}_1 z_1, x^{i+1} = (z_1, y_2)$. In this case we are observing the transition $(x_1, x_2) \xrightarrow{\sigma_{i+1}}_H (z_1, y_2)$ which does not exist in G. By including τ transitions however we have $(x_1, x_2) \xrightarrow{\tau^+}_G (y_1, x_2) \xrightarrow{\sigma_{i+1}}_G (z_1, y_2)$, thus $(x^i, x_T^i) = (x_1, x_2, x_T^i) \overset{P(\sigma_{i+1})}{\Longrightarrow} (z_1, y_2, x_T^i) = (x^{i+1}, x_T^{i+1})$ in $G||T$.

Since all possible cases yield in a possible path in $G||T$ to the target state and this holds for an arbitrary transition on the path $(*)$, it follows that $(x, x_T) = (x^0, x_T^0) \overset{P(\sigma_1)}{\Longrightarrow} (x^1, x_T^1) \overset{P(\sigma_2)}{\Longrightarrow} \cdots \overset{P(\sigma_n)}{\Longrightarrow} (x^n, x_T^n) = (y, y_T)$, and thus $(x, x_T) \overset{P(s)}{\Longrightarrow} (y, y_T)$ in $G||T$. □

Lemma 3. If $(w, w_T) \overset{t}{\Rightarrow} (x, x_T)$ in $G||T$ then there exist (y, y_T) and $t' \in \Sigma^*$ such that

- $(x, x_T) \overset{\varepsilon}{\Rightarrow} (y, y_T)$ in $G||T$.
- $(w, w_T) \overset{t'}{\Rightarrow} (y, y_T)$ in $H||T$.
- $\forall \sigma : \sigma \in t \iff \sigma \in t'$.

Proof. Let $t = \sigma_1 \ldots \sigma_n$ and $(w, w_T) = (x^0, x_T^0) \xrightarrow{\sigma_1} (x^1, x_T^1) \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} (x^n, x_T^n) = (x, x_T)$ in $G||T$. It is shown by induction on $i = 0 \ldots n$ that there exist paths η_i and θ_i such that $t = \eta_0 \circ \theta_0$ and

1. $\forall \sigma \in \Sigma : \sigma \in \eta_i \circ \theta_i \iff \sigma \in t$.
2. θ_i is a path in $G||T$.
3. η_i is a path in $H||T$.
4. $(x, x_T) \stackrel{\varepsilon}{\Rightarrow} (x', x_T{}')$ where $(x', x_T{}')$ is the final state of θ_i in $G||T$.
5. $|\theta_i| \leq n - i$.

Base case $i = 0$, let $\eta_0 = (x^0, x_T^0)$, $\theta_0 = t$. Clearly $\eta_0 \circ \theta_0 = t$.

1. $\eta_0 \circ \theta_0 = t$ contains exactly all of the events of t.
2. $\theta_0 = t$ is a path in $G||T$ by assumption.
3. $\eta_0 = (x^0, x_T^0)$ is a path in $H||T$ as every state forms a path.
4. $(x^{n\prime}, x_T^{n\prime}) = (x^n, x_T^n)$ in this instance and every state can reach itself silently.
5. $|\theta_0| = |t| = n = n - 0 = n - i$.

Assume this holds for i, now consider $i + 1$. Let $x^i = (x_1^i, x_2^i)$. The following cases arise when selecting η_{i+1} and θ_{i+1}:

- $\sigma_{i+1} \in \Sigma_1 \cup \Sigma_2 \vee (\sigma_{i+1} = \tau \wedge (x_1 \stackrel{\tau}{\not\rightarrow} \vee x_2 \stackrel{\tau}{\not\rightarrow}))$. These conditions make the transition fall into one of three categories for the transition relation in H. It could be the case that $\sigma_{i+1} \in \Sigma_1 \cap \Sigma_2$. If so then it cannot be the case that $\sigma_{i+1} = \tau$ as that would lead to both $x_1 \stackrel{\tau}{\rightarrow} \wedge x_2 \stackrel{\tau}{\rightarrow}$, so the transition exists in H in this case. If $\sigma_{i+1} = \tau$ it is also the case that at least one of x_1 and x_2 disable τ, leading to the transition existing in H by the second or third rule. The only remaining case is that $\sigma_{i+1} \in \Sigma_1 \setminus \Sigma_2$ or $\sigma_{i+1} \in \Sigma_2 \setminus \Sigma_1$, both of which yield that the transition exists in H also by rules two or three. Since all possible cases lead to the same transition being in H, we can let $\eta_{i+1} = \eta_i \circ (x^i, x_T^i) \xrightarrow{\sigma_{i+1}} (x^{i+1}, x_T^{i+1})$ and $\theta = (x^{i+1}, x_T^{i+1}) \xrightarrow{\sigma_{i+2}} \cdots \xrightarrow{\sigma_n} (x^n, x_T^n)$. This just shifts one event from θ_i to η_i, so we have $\eta_{i+1} \circ \theta_{i+1} = \eta_i \circ \theta_i$.

1. $\eta_{i+1} \circ \theta_{i+1} = \eta_i \circ \theta_i$ contains exactly all the events of t by inductive assumption.
2. θ_{i+1} is a path in $G||T$ as θ_i is a path in $G||T$ by assumption and θ_{i+1} was obtained by simply removing one event from θ_i.
3. η_{i+1} is a path in $H||T$ as η_i is a path in $H||T$ by inductive assumption and it has been shown that $(x^i, x_T^i) \xrightarrow{\sigma_{i+1}} (x^{i+1}, x_T^{i+1})$ exists in $H||T$.
4. The final state of θ_{i+1} is the same as the final state of θ_i, which is reachable silently from (x^n, x_T^n) by inductive assumption.
5. $|\theta_{i+1}| = |\theta_i| - 1 = n - i - 1 = n - (i + 1)$

- $(\sigma_{i+1} \in \Sigma_T \setminus (\Sigma_1 \cup \Sigma_2)) \vee (\sigma_{i+1} = \tau \wedge x_T^i \neq x_T^{i+1})$. In this case the transition belongs to the test T and as such will still exist in $H||T$, as such the choices for η_{i+1} and θ_{i+1} are the same as above. Let $\eta_{i+1} = \eta_i \circ (x^i, x_T^i) \xrightarrow{\sigma_{i+1}} (x^{i+1}, x_T^{i+1})$ and $\theta = (x^{i+1}, x_T^{i+1}) \xrightarrow{\sigma_{i+2}} \cdots \xrightarrow{\sigma_n} (x^n, x_T^n)$.

  1. $\eta_{i+1} \circ \theta_{i+1} = \eta_i \circ \theta_i$ contains exactly all the events of t by inductive assumption.
  2. θ_{i+1} is a path in $G||T$ as θ_i is a path in $G||T$ by assumption and θ_{i+1} was achieved by simply removing one event from θ_i.
  3. η_{i+1} is a path in $H||T$ as η_i is a path in $H||T$ by inductive assumption and it has been shown that $(x^i, x_T^i) \xrightarrow{\sigma_{i+1}} (x^{i+1}, x_T^{i+1})$ exists in $H||T$.
  4. The final state of θ_{i+1} is the same as the final state of θ_i, which is reachable silently from (x^n, x_T^n) by inductive assumption.
  5. $|\theta_{i+1}| = |\theta_i| - 1 = n - i - 1 = n - (i + 1)$

- $x_1 \xrightarrow{\tau}, x_2 \xrightarrow{\tau}, \sigma_{i+1} = \tau$ and $x_1^i \neq x_1^{i+1}$. Let $j > i$ be the smallest index such that $j = n$ or $\sigma_{j+1} \neq \tau$ or $x_1^j = x_1^{j+1}$. Here we have one or more silent transitions from A_1, depending on the event that follows this sequence of τ gives rise to the following cases:

– $\sigma_{j+1} \in \Sigma_1$, then we have the path $(x_1^i, x_2^i, x_T^i) \xrightarrow{\tau=\sigma_{i+1}} (x_1^{i+1}, x_2^{i+1}, x_T^i) \xrightarrow{\tau=\sigma_{i+2}} \cdots \xrightarrow{\tau=\sigma_j} (x_1^j, x_2^j, x_T^j) \xrightarrow{\sigma_{j+1}} (x_1^{j+1}, x_2^{j+1}, x_T^i)$. Since $\sigma_{j+1} \in \Sigma_1$ and $(x_1^j, x_2^j, x_T^j) \xrightarrow{\sigma_{j+1}} (x_1^{j+1}, x_2^{j+1}, x_T^i)$ this means that $x_2^i \xrightarrow{\sigma_{j+1}} x_2^{j+1}$ as the events from Σ_1 that preceded σ_{j+1} do not affect x_2^i. We also have $x_1^i \xrightarrow{\tau^+} x_1^j \xrightarrow{\sigma_{j+1}} x_1^{j+1}$, which means that $(x^i, x_T^i) \xrightarrow{\sigma_{j+1}} (x^{j+1}, x_T^{j+1})$ exists in $H||T$ by rule 5. Now let $\eta_{i+1} = \eta_i \circ (x^i, x_T^i) \xrightarrow{\sigma_{j+1}} (x^{j+1}, x_T^{j+1})$ and $\theta_{i+1} = (x^{j+1}, x_T^{j+1}) \xrightarrow{\sigma_{j+2}} \cdots \xrightarrow{\sigma^n} (x^n, x_T^n)$. This has removed all of the τ transitions between σ_i and $\sigma_{j+1}$and has left the path otherwise unaltered.

  1. $\eta_{i+1} \circ \theta_{i+1}$ contains exactly the events of $\eta_i \circ \theta_i$, minus the removed τ events. Since $\eta_i \circ \theta_i$ contained only the events of t by inductive assumption, it holds that $\eta_{i+1} \circ \theta_{i+1}$ also does.
  2. θ_{i+1} is a path in $G||T$ as θ_i is a path in $G||T$ by assumption and θ_{i+1} was achieved by just removing transitions from θ_i.
  3. η_{i+1} is a path in $H||T$ as η_i is a path in $H||T$ by inductive assumption and it has been shown that $(x^i, x_T^i) \xrightarrow{\sigma_{j+1}} (x^{j+1}, x_T^{j+1}$ exists in $H||T$.
  4. The final state of θ_{i+1} is the same as the final state of θ_i, which is reachable silently from (x^n, x_T^n) by inductive assumption.
  5. $|\theta_{i+1}| = |\theta_i| - (j-i) \leq n - i - (j-i) = n - j \leq n - (i+1)$ since $i < j$ and $0 \leq i \leq n$.

– $(\sigma_{j+1} \in (\Sigma_2 \cup \Sigma_T) \setminus \Sigma_1) \ \vee \ (\sigma_{j+1} = \tau \ \wedge \ x_T^j \neq x_T^{j+1})$. Here σ_{j+1} is either a normal event from A_2, a normal event from T, or a τ transition from T. We can observe the fact that σ_{j+1} is independent of each σ_k with $i < k \leq j$ and as such we have $(x^k, x_T^k) \xrightarrow{\sigma_{j+1}} (y^k, y_T^k)$ and $(y^k, y_T^k) \xrightarrow{\tau} (y^{k+1}, y_T^{k+1})$. This means that path $(x^i, x_T^i) \xrightarrow{\sigma_{j+1}} (y^i, y_T^i) \xrightarrow{\sigma_{i+1}} \cdots \xrightarrow{\sigma_{j-1}} (y^{j-1}, y_T^{j-1}) \xrightarrow{\sigma_j} (x^{j+1}, x_T^{j+1})$ exists in both $H||T$ and $G||T$, so let $\eta_{i+1} = \eta_i \circ (x^i, x_T^i) \xrightarrow{\sigma_{j+1}} (y^i, y_T^i)$ and $\theta_{i+1} = (y^i, y_T^i) \xrightarrow{\sigma_{i+1}} \cdots \xrightarrow{\sigma_{j-1}} (y^{j-1}, y_T^{j-1}) \xrightarrow{\sigma_j} (x^{j+1}, x_T^{j+1}) \xrightarrow{\sigma_{j+2}} \cdots \xrightarrow{\sigma_n} (x^n, x_T^n)$. Here we have moved the independent σ_2 transition to the beginning of the θ_i path and transferred it to the end of the η_i path. The θ_{i+1} path then consists of two parts, a path

j in length that is parallel to θ_i, and the rest of θ_i once the last event of the parallel path appears on θ_i.

1. $\eta_{i+1} \circ \theta_{i+1}$ contains exactly the events of $\eta_i \circ \theta_i$ with just one event moved to slightly earlier in sequence. Since $\eta_i \circ \theta_i$ contained only the events of t by inductive assumption, it holds that $\eta_{i+1} \circ \theta_{i+1}$ also does.
2. θ_{i+1} is a path in $G||T$ as it has been shown that $(y^i, y_T^i) \xrightarrow{\sigma_{i+1}} \cdots \xrightarrow{\sigma_{k-2}} (y^{k-2}, x_T^{k-2}) \xrightarrow{\sigma_{k-1}} (x^{j+1}, x_T^{j+1})$ is a path in $G||T$ and $(x^{j+1}, x_T^{j+1}) \xrightarrow{\sigma_{j+2}} \cdots \xrightarrow{\sigma_n} (x^n, x_T^n)$ is a part of θ_i which is a path in $G||T$ by inductive assumption.
3. η_{i+1} is a path in $H||T$ as η_i is a path in $H||T$ by inductive assumption and it has been shown that $(x^i, x_T^i) \xrightarrow{\sigma_{j+1}} (y^i, y_T^i$ exists in $H||T$.
4. The final state of θ_{i+1} is the same as the final state of θ_i, which is reachable silently from (x^n, x_T^n) by inductive assumption.
5. $|\theta_{i+1}| = |\theta_i| - 1 \geq n - i - 1 = n - (i + 1)$.

– $\sigma_{j+1} = \tau \wedge x_2^j \neq x_2^{j+1}$, then similar to the previous case, σ_{j+1} is independent of each σ_k with $i < k \leq j$. This means that $(x_1^i, x_2^i, x_T^i) \xrightarrow{\tau=\sigma_{i+1}} (x_1^{i+1}, x_2^i, x_T^i) \xrightarrow{\tau=\sigma_{j+1}} (x_1^{i+1}, x_2^{j+1} x_T^i) \xrightarrow{\tau=\sigma_{i+2}}$
$(x_1^{i+2}, x_2^{j+1}, x_T^i) \xrightarrow{\tau=\sigma_{i+3}} \cdots \xrightarrow{\tau=\sigma_j} (x_1^{j+1}, x_2^{j+1}, x_T^i = x_T^{j+1}) = (x^{j+1}, x_T^{j+1})$ exists in $G||T$. Notice here that we have $x_1^i \xrightarrow{\tau} x_1^{i+1}$ and $x_2^i \xrightarrow{\tau} x_2^{j+1}$ and as such using $\rightarrow_H$ we have that $(x^i, x_T^i) = (x_1^i, x_2^i, x_T^i) \xrightarrow{\tau}$ $(x_1^{i+1}, x_2^{j+1}, x_T^i) = (y^{i+1}, y_T^{i+1})$ exists in $H||T$. Let $\eta_{i+1} = \eta_i \circ$ $(x^i, x_T^i) \xrightarrow{\tau} (y^{i+1}, y_T^{i+1})$ and $\theta_{i+1} = (y^{i+1}, y_T^{i+1}) \xrightarrow{\sigma_{i+2}} \cdots \xrightarrow{\sigma_{j-1}}$ $(y^{j-1}, y_T^{j-1}) \xrightarrow{\sigma_j} (x^{j+1}, x_T^{j+1})$
$\xrightarrow{\sigma_{j+2}} \cdots \xrightarrow{\sigma_n} (x^n, x_T^n)$. Here we have moved the σ_{j+1} transition to the beginning of the θ_i path and used the fact that a τ transition from $\rightarrow_1$ and a τ transition from $\rightarrow_2$ in succession are combined into a single τ transition in H to create the τ transition at the end of η_{i+1}, where the θ_{i+1} construction works in the same way as for the $\sigma_{j+1} \in \Sigma_2$ case.

1. $\eta_{i+1} \circ \theta_{i+1}$ contains exactly the events of $\eta_i \circ \theta_i$ with just one

less τ event. Since $\eta_i \circ \theta_i$ contained only the events of t by inductive assumption, it holds that $\eta_{i+1} \circ \theta_{i+1}$ also does.

2. θ_{i+1} is a path in $G||T$ as it has been shown that $(y^{i+1}, y_T^{i+1}) \xrightarrow{\sigma_{i+2}} \cdots \xrightarrow{\sigma_k} (x^{j+1}, x_T^{j+1})$ is a path in $G||T$ and $(x^{j+1}, x_T^{j+1}) \xrightarrow{\sigma_{j+2}} \cdots \xrightarrow{\sigma_n} (x^n, x_T^n)$ is a part of θ_i which is a path in $G||T$ by inductive assumption.
3. η_{i+1} is a path in $H||T$ as η_i is a path in $H||T$ by inductive assumption and it has been shown that $(x^i, x_T^i) \xrightarrow{\tau} (y^{i+1}, y_T^{i+1}$ exists in $H||T$.
4. The final state of θ_{i+1} is the same as the final state of θ_i, which is reachable silently from (x^n, x_T^n) by inductive assumption.
5. $|\theta_{i+1}| = |\theta_i| - 2 \leq n - i - 2 = n - (i + 2) \leq n - (i + 1)$.

– $j = n$. Here all of the remaining transitions are silent transitions from A_1. Recall that $x_1^i \xrightarrow{\tau}_1$ and $x_2^i \xrightarrow{\tau}_2$. Let $x_2^i \xrightarrow{\tau}_2 x_2^{i\prime}$, this means that the paths $(x_1^i, x_2^i, x_T^i) \xrightarrow{\sigma_{i+1}} (x_1^{i+1}, x_2^i, x_T^i) \xrightarrow{\sigma_{i+2}} \cdots \xrightarrow{\sigma_n} (x_1^n, x_2^i, x_T^i) \xrightarrow{\tau} (x_1^n, x_2^{i\prime}, x_T^i) = (y^n, y_T^n)$ and $(x_1^i, x_2^i, x_T^i) \xrightarrow{\tau} (x_1^i, x_2^{i\prime}, x_T^i) \xrightarrow{\sigma_{i+1}} (x_1^{i+1}, x_2^{i\prime}, x_T^i) \xrightarrow{\sigma_{i+2}} \cdots \xrightarrow{\sigma_{n-1}} (x_1^{n-1}, x_2^{i\prime}, x_T^i) \xrightarrow{\sigma_n} (x_1^n, x_2^{i\prime}, x_T^i) = (y^n, y_T^n)$ exist in $G||T$. Accordingly using $\rightarrow_H$ we have that $(x_1^i, x_2^i, x_T^i) \xrightarrow{\tau} (x_1^{i+1}, x_2^{i\prime}, x_T^i) = (y^{i+1}, y_T^{i+1}) \xrightarrow{\sigma_{i+2}} (y^{i+1}, y_T^{i+1}) \xrightarrow{\sigma_{i+3}} \cdots \xrightarrow{\sigma_n} (y^n, y_T^n)$ exists in $H||T$. Let $\eta_{i+1} = \eta_i \circ (x^i, x_T^i) \xrightarrow{\tau} (y^{i+1}, y_T^{i+1})$ and $\theta_{i+1} = (y^{i+1}, y_T^{i+1}) \xrightarrow{\sigma_{i+2}} \cdots \xrightarrow{\sigma_n} (y^n, y_T^n)$. Here we have constructed a parallel path to a state that is reachable from (x^n, x_T^n) using a silent transition by using the fact that a τ transition from $\rightarrow_1$ and a τ transition from $\rightarrow_2$ in succession are combined into a single τ transition in H.

1. $\eta_{i+1} \circ \theta_{i+1}$ contains exactly the events of $\eta_i \circ \theta_i$ with just one less τ event. Since $\eta_i \circ \theta_i$ contained only the events of t by inductive assumption, it holds that $\eta_{i+1} \circ \theta_{i+1}$ also does.
2. θ_{i+1} is a path in $G||T$ as it has been shown that $(y^{i+1}, y_T^{i+1}) \xrightarrow{\sigma_{i+2}} \cdots \xrightarrow{\sigma_n} (y^n, y_T^n)$ is a path in $G||T$.
3. η_{i+1} is a path in $H||T$ as η_i is a path in $H||T$ by inductive assumption and it has been shown that $(x^i, x_T^i) \xrightarrow{\tau} (y^{i+1}, y_T^{i+1}$

exists in $H||T$.

4. The final state of θ_{i+1} has been shown to be reachable silently from the final state of θ_i, which is reachable silently from (x^n, x_T^n) by inductive assumption.
5. $|\theta_{i+1}| = |\theta_i| - 1 \leq n - i - 1 = n - (i + 1)$.

Since every possible case yields a path which satisfy conditions $1 \ldots 5$, when $i = n$ we will have $|\theta_i| = 0$ and $|\eta_i| = |\eta_i \circ \theta_i|$, meaning that η_i is the entire path. From condition 3 we know that this means the entire path is a path in $H||T$, and from condition 4 we know that this path ends in a state that is reachable silently from the original end state of t. With these conditions satisfied then, we have our result. □

Theorem 7. $G \simeq_{conf} H$

Proof. The claim is equivalent to saying that for any test T, if $G||T$ is nonblocking then so is $H||T$ and vice versa.

First assume that $G||T$ is nonblocking, we must show that $\forall s, x, x_T : H||T \stackrel{s}{\Rightarrow} (x, x_T), \exists t : (x, x_T) \stackrel{t\omega}{\Rightarrow} inH||T$. Assume $H||T \stackrel{s}{\Rightarrow} (x, x_T)$, by Lemma 2 we then have $G||T \stackrel{P_\tau(s)}{\Longrightarrow} (x, x_T)$. It remains to be shown that $\exists t : (x, x_T) \stackrel{t'\omega}{\Rightarrow}$ in $H||T$. Since we know that $G||T$ is nonblocking, we know that $\exists t : (x, x_T) \stackrel{t\omega}{\rightarrow}$. Let $t\omega = u$. By Lemma 3 we know that there exists the path u' where $(x, x_T) \stackrel{u'}{\Rightarrow}$ in $H||T$ and $\forall \sigma : \sigma \in u \iff \sigma \in u'$. Since $\omega \in u$ this implies that $\omega \in u'$. We now have $H||T \stackrel{s}{\Rightarrow} (x, x_T) \stackrel{u'}{\Rightarrow}$, and since $\omega \in u'$, it follows that $H||T$ is nonblocking.

Now assume that $H||T$ is nonblocking and $G||T \stackrel{s}{\Rightarrow} (x, x_T)$, we must show that $(x, x_T) \stackrel{t\omega}{\Rightarrow}$ in $G||T$. By Lemma 3 we have $H||T \stackrel{s'}{\Rightarrow} (y, y_T)$ with $(x, x_T) \stackrel{\varepsilon}{\Rightarrow} (y, y_T)$ in $G||T$. Since $H||T$ is nonblocking we have $(y, y_T) \stackrel{t\omega}{\Rightarrow}$ in $H||T$. We then have by Lemma 2 that $(y, y_T) \stackrel{t\omega}{\Rightarrow}$ in $G||T$. Combined with the previous result we then have $(x, x_T) \stackrel{\varepsilon}{\Rightarrow} (y, y_T) \stackrel{t\omega}{\Rightarrow}$ in $G||T$, therefore $G||T$ in nonblocking.

Since we have $G||T$ nonblocking $\implies$ $H||T$ nonblocking and $H||T$ nonblocking $\implies$ $G||T$ nonblocking, we have our result that $G \simeq_{conf} H$. $\square$

4.3 Conclusions

At the time of writing this thesis a fully functional implementation of the algorithms described in this chapter has not yet been realised. As such we are unable to provide effective results with which to draw meaningful conclusions from. Instead in this section we will offer conclusions based on what is understood of the processes involved and how we would expect the developments outlined in this chapter to impact them.

Since the silent continuation with independence rule is based on the silent continuation rule, we should expect to see similar results from the two methods. The main difference with the new rule however is that it is performed as the synchronous product is being created, rather than identifying conflict equivalent states in the component automata. This should mean that using the silent continuation with independence rule allows for faster verification while achieving similar state-space reductions, as the time spent identifying the conflict equivalent states in the component automata can be avoided. There is obviously computational overhead involved in performing this process as the synchronous process is being created however. Since the only states of interest when performing this process are global states with several τ enabled, and these states would be found during a regular synchronous product builder anyway, then no extra time is spent searching for these states. Finding the τ successors does amount to additional comutation, but this can be compared to the time spent searching the component automata when using the original rule, while doing the search during the synchronous composition creation is effetively performing the search on all of the component automata at once.

It will be of great interest to see how the different abstractions compare, and how the abstraction introduced in this chapter interacts with the other

reduction rules for compositional verification that were not explicitly mentioned.

Chapter 5

Conclusions

New partial order reduction model verifiers for nonblocking and controllability that take advantage of several optimisations have been presented. It has been shown that these verifiers perform substantially better than the previous offering given in [17], with run times significantly lower particularly when verifying models with a large number of events. We have identified several key areas where the partial order reduction method excels and struggles, and this information should be able to be used to further optimise the process or develop different implementations.

A new abstraction rule for compositional verification has also been developed. While there are not yet any experimental results to verify how successful this abstraction will be, it has been shown that as the rule is applied during synchronous composition, it is in a position to take advantage of the information awarded by several component automata at once, which is not something that any other reduction rules are currently capable of.

Further work in these areas could include but are not limited to further investigating how to best compute ample sets by expermenting with different orderings of the events in the enabled event sets. Analysing specifically the models that perform particularly well or that struggle to achieve reduction where we might expect it, should give valuable insight into how to best organise these event sets. Another improvement could be to combine the two

methods discussed in this thesis, so that partial order reduction becomes a part of the compositional verification process. As well as that other reduction techniques could be used on combination with these methods such as using symbolic model checking [14] together with compositional or partial order reduction verification. Symbolic model checking with partial order reduction [2] would involve iterpreting the independence relation so that it could be expressed in OBDDs and then using that to determine a way to conduct a search of a symbolic state space while under some restrictions imposed by the independence relation.

Chapter 6

Bibliography

[1] Knut Åkesson, Martin Fabian, Hugo Flordal, and Robi Malik. Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems. In *Proceedings of the 8th International Workshop on Discrete Event Systems, WODES'06*, pages 384–385, Ann Arbor, MI, USA, July 2006.

[2] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Proceedings of 9th International Conference on Computer Aided Verification, CAV '97*, volume 1254 of *LNCS*, pages 340–351. Springer, 1997.

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

[4] Bertil A. Brandin, Robi Malik, and Petra Malik. Incremental verification and synthesis of discrete-event systems guided by counter-examples. *IEEE Transactions on Control Systems Technology*, 12(3):387–401, May 2004.

[5] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, September 1999.

[6] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[7] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[8] Hugo Flordal and Robi Malik. Compositional verification in supervisory control. *SIAM Journal of Control and Optimization*, 48(3):1914–1938, 2009.

[9] Rachel Francis. An implementation of a compositional approach for verifying generalised nonblocking. Working Paper 04/2011, Department of Computer Science, University of Waikato, Hamilton, New Zealand, 2011.

[10] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. PhD thesis, Université de Liège, Faculté des Sciences Appliqués, 1995.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[12] R. Malik and R. Mühlfeld. A case study in verification of UML statecharts: the PROFIsafe protocol. *Journal of Universal Computer Science*, 9(2):138–151, February 2003.

[13] Robi Malik and Reinhard Mühlfeld. Testing the PROFIsafe protocol using automatically generated test cases based on a formally verified model. Technical report, Siemens AG, Corporate Technology, Software and Engineering 1, Munich, Germany, 2002.

[14] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[15] KorSys Project. http://www4.in.tum.de/proj/korsys/.

[16] Peter J. G. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[17] Adrian Shaw. Partial order reduction in discrete event systems. Honours project report, Department of Computer Science, University of Waikato, 2013.

[18] Supremica. `www.supremica.org`. The official website for the Supremica project.

[19] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[20] Simon Ware and Robi Malik. The use of language projection for compositional verification of discrete event systems. In *Proceedings of the 9th International Workshop on Discrete Event Systems, WODES'08*, pages 322–327, Göteborg, Sweden, May 2008.

[21] W. M. Wonham. Supervisory control of discrete-event systems. Systems Control Group, http://www.control.utoronto.edu/, Department of Electrical Engineering, University of Toronto, Ontario, Canada, 2007.

[22] W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete event systems. *Mathematics of Control, Signals and Systems*, 1(1):13–30, January 1988.