Working Paper Series
ISSN 1177-777X

AN IMPLEMENTATION OF A COMPOSITIONAL APPROACH FOR VERIFYING GENERALISED NONBLOCKING

Rachel Francis

Working Paper: 04/2011
27 April 2011

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Abstract

Generalised nonblocking is a property of discrete-event systems which verifies liveness. It was introduced to overcome the weaknesses of standard nonblocking. Verifying generalised nonblocking of real-world models often involves exploring state-spaces which will exceed available memory. A compositional verification approach has been developed to achieve verification for models of a much larger size. For this project, we have developed the first implementation for compositionally verifying generalised nonblocking. In addition, we have experimented with the techniques used in compositional verification, and analysed their performance. Our algorithm has successfully verified a large set of industrial-size models, including at least one large model which had not been verified before.

Contents

1 Introduction

It is desirable to verify that real-world systems *can* complete particular tasks. For example, in a factory we would want to be sure once a process is started, such a process can eventually be completed. The state the factory is in after the process completes is usually referred to as a terminal state. Verifying that terminal states are reachable after something particular has occurred is important for many systems to ensure correct operation and reliability. This verification helps recognise a system which will not operate as one would expect, such as a process in the factory starting and later the factory entering a state where that process can never complete.

Real-life systems can be modelled as a set of finite-state automata which are designed to behave the same as the components of the system. The interaction between components is replicated by automata running in parallel with each other (Ramadge & Wonham, 1989). Once the system is modelled correctly, we are able to verify whether the model is nonblocking. Nonblocking is a property which has two variations; standard nonblocking is satisfied when all reachable can reach a terminal state (Ramadge & Wonham, 1989), generalised nonblocking extends this standard property. For generalised nonblocking, only states where some precondition has occurred must be capable of reaching a terminal state (Leduc & Malik, Generalised Nonblocking, 2008).

The standard approach for verifying nonblocking of a model is completed in two main steps. The first step is building the synchronous product of the entire model; this involves composing all automata contained by the model in to one single finite-state automaton. The second step for verifying nonblocking is to apply a model checking algorithm to the composed model in order to verify the property in question. Unfortunately, building the synchronous product for a real-world model has two major downsides. The first shortcoming being the time involved for complete construction – the construction process takes time proportional to the number of reachable states in the composed system, which increases exponentially by the number of automata in the whole system. The second shortcoming is known as the state space explosion problem (Berard, et al., 2001). This problem occurs when the number of states of the synchronous product is too large to fit in memory.

A compositional verification approach has been developed in an attempt to alleviate the above disadvantages (Flordal & Malik, 2009). Compositional verification involves reducing the size of a model, and ensuring that the simplified model contains exactly the same faults as its original. Compositional algorithms for verifying standard nonblocking do exist and are available in the tools Supremica (Akesson, Fabian, Flordal, & Malik, 2006), (Supremica, 2010) and VALID (Brandin, 2000). However, no existing compositional implementations handle generalised nonblocking. For this project, we developed the first implementation for compositional verification of generalised nonblocking. The algorithm is written in Java and uses an explicit representation of the automata internally. We do not use any symbolic representation (Berard, et al., 2001). The reason for this being that we expect there would not be much to gain, since the intention of compositional verification is that we will not often encounter automata of more than a few thousand states. In fact, prior to this project no implementations at all existed for verifying generalised nonblocking, even using standard approaches. Therefore, in the early stages of this project we also extended a conflict checker which used standard approaches to verify nonblocking, so it could also handle generalised nonblocking verification.

The compositional approach for verifying generalised nonblocking is to select and compose a few automata of the model and then apply conflict-preserving abstraction rules to significantly reduce the model's size. The abstracted automaton then replaces the original automata it was composed from in the model and the process is repeated, usually many times. In theory, following this process allows any model to be reduced to only one automaton with a single state and no transitions, however it is not usually practicable due to time and memory constraints. A model is simplified by our implementation as far as desired, and then has generalised nonblocking verified by model checkers which already exist. The conflict checkers we use after composition use a standard approach for verification, which includes explicitly constructing the synchronous product state space.

Counterexamples provide diagnostics for why a model is blocking. After simplifying a model and using a standard approach conflict checker, if the model is blocking, a counterexample is provided which is correct for the *simplified* model. Therefore, our implementation needs to expand a counterexample for a simplified model into an equivalent counterexample which is accepted by the original model.

The nature of the compositional approach means the user can configure several features in an attempt to get the fastest possible result. Thus, the most interesting part of this project is evaluating the most effective way of configuring compositional verification. These configurations include the method for selecting automata to compose and the ordering of the abstraction rules. Each of the abstraction rules has been evaluated in terms of what reduction in automata size they lead to and how they influence the efficiency of the overall time taken to check whether the system satisfies generalised nonblocking.

Our implementation has been tested thoroughly using a large existing set of models. This set contains models of varying sizes, some of which are very large and replicate the behaviour of real world systems. Using our algorithm we have successfully verified a large set of models, including at least one large model which had not been verified before. The option to verify generalised nonblocking using our compositional implementation has been integrated in the Supremica toolkit.

This dissertation comprises the chapters described below.

Chapter 2 discusses some of the fundamental theory required to understand this dissertation. Chapter 3 discusses the compositional approach and how it has been implemented for this project. In chapter 4, the results of experimenting with our implementation are evaluated. Chapter 5 presents the overall conclusions for compositional verification and this project.

2 Preliminaries

This chapter presents a set of definitions for terms used regularly throughout this dissertation; it is intended to aid the understanding of the report content.

2.1 Multi-coloured Finite-state Automata

An automaton is a simple means for *modelling* a system's possible behaviour, while event sequences and languages *describe* the same system's behaviour (Ramadge & Wonham, 1989). Events are members of a finite alphabet $\sum$. In conjunction with this alphabet of events we use a *silent event* $\tau \notin \sum$, with the notation $\sum_\tau = \sum \cup \{\tau\}$. $\sum^*$ denotes the set of all finite strings of the form $\sigma_1\ \sigma_2 \ldots\ \sigma_n$ of events from $\sum$, including the *empty string* ϵ.

Finite-state automata have *marking propositions* associated with certain states. Traditionally, finite-state automata have only one marking proposition which identifies a state where operation can safely terminate. Multi-coloured automata extend the traditional marking concept to using multiple marking propositions simultaneously in a model. The addition of other markings allows particular states to show a precondition has been satisfied.

Nondeterminism is a crucial attribute of an automaton for the abstraction techniques used in the implementation presented in this report. The following definition is introduced in (Leduc & Malik, 2008).

Definition A nondeterministic multi-coloured finite-state automaton is a tuple $A = (\sum, \prod, X, \rightarrow, X^\circ, \Xi)$, where $\sum$ is a finite set of events, $\prod$ is a finite set of propositions, X is a finite set of states, $\rightarrow\ \subseteq X \times \sum_\tau \times X$ is the state transition relation, $X^\circ \subseteq X$ is the set of initial states, and $\Xi : \prod \rightarrow 2^X$ defines the set of marked states for each proposition in $\prod$.

An automaton is depicted by labelled arrows representing transitions and labelled circles representing states. Figure 2.1 shows an example of a simple automaton. The circle labelled S0 represents a state. At least one state will always have an arrow leading in to it (from nowhere) which signifies that that state is the initial state of the automaton; this is S0 in Figure 2.1. States can have marking propositions associated with them. The multi-coloured automata studied for this project can have two marking propositions. The traditional default ω-marking proposition is used for states where operation can safely terminate, and an α-marking indicates a state which satisfies some precondition. S1 depicts a *terminal* state by a circle coloured black and S2 is coloured grey to illustrate it has an α-marking.

The arrow labelled e2 shows a transition between the source state S2 and the target state S1. A self-loop transition is shown labelled e3 with S2 as the source and target state. The labels associated with transitions represent the event required to make that particular transition between two states. Transitions can be expressed as $s \overset{e}{\to} t$, where s is the source state, t is the target state and e is the event name required to move from source to target.

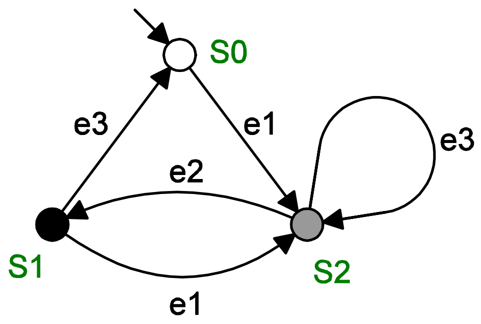

Figure 2.1: A simple multi-coloured finite-state automaton.

A *model* of a system represents a complete system and will usually consist of many automata. The event alphabet of a model consists of each unique event and proposition used by the model's automata. If an automaton of a model does not contain some marking proposition which is in the model's event alphabet, all states of that automaton are regarded as being implicitly marked with that proposition.

2.2 Synchronous Product

To correctly portray a discrete-event system all automata must run in parallel. Modelling the parallel execution of multiple automata is done by *synchronous composition*, using a lock-step approach as introduced in (Hoare, 1985). Building the synchronous product of a system composes all automata in to one automaton.

For the purpose of this report a *model* is a collection of finite-state automata, which together model a system. Each automaton of a model will usually contain events in their alphabet which other automata in the model also use; these events are called *shared events*. Shared events must be executed by all automata synchronously. In contrast, each automaton may also have *local events,* events which are in that automaton's event alphabet but are *not* shared with any other automaton and therefore they are executed independently.

Building the synchronous product assumes that all automata composed share the same event and proposition alphabet. However, often not all automata of a model will share all the same event alphabets, and allowances must be made before composing the model. We do this by considering for each automaton, all events not in that automaton's alphabet which are used by other automata to be composed with, are implicitly part of that excluded automaton's alphabet. We consider all states of an automaton without an event as having a self-looped transition $x \overset{e}{\rightarrow} x$ for all states $x \in X$. This allows the event e to occur in any state of the automaton, without affecting the system's state.

Example In Figure 2.2 the event *e3* is part of A_1's alphabet, but not A_2's alphabet. Thus, *e3* is *local* to A_1, this is also the case for *e4* in A_2. Both other events (*e1* and *e2*) are members of both *A1* and *A2*'s event alphabets, thus they are *shared* events for this model.

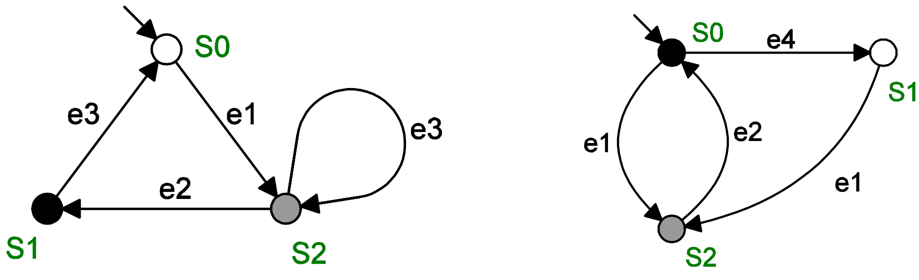

Figure 2.2: Automaton A_1 on the left, automaton A_2 on the right.

Example The synchronous product of A_1 and A_2, which we denote by $A_1 \parallel A_2$, is shown in Figure 2.3. The labels for the states are the corresponding state names of A_2 appended to A_1's state name. State names are irrelevant, but easier to understand when a naming convention like this is used. Marking propositions in a synchronously composed automaton can only apply to a composed state, of which all the automata involved had their composed state marked. For A_1 the only ω-marked state is S1 and the same for S0 of A_2. Therefore, the only state of $A_1 \parallel A_2$ which can be ω-marked is the composition of both these states, S1.S0. The same concept applies to states which are the initial states of $A_1 \parallel A_2$. The transitions of $A_1 \parallel A_2$ reflect the possible transitions from each state of the automata in the composition. $A_1 \parallel A_2$'s initial state S0.S0 can only have the same outgoing transitions as S0 in A_1 and S0 in A_2. A_1's state S0 has en outgoing *e1*-transition, as does S0 of A_2. Therefore $A_1 \parallel A_2$ has an outgoing *e1*-transition to a state composed of the target states for the individual transitions in A_1 and A_2, this state is S2.S2. S0.S0 also has an outgoing *e4*-transition to S0.S1. This shows that A_1 did not transition to another state, which is correct since *e4* is local to A_2. Behind the scenes we implicitly consider S0 in A_1 as having an *e4*-self loop since *e4* is not in A_1's event alphabet and for multiple automata to be synchronised they must be of the same event alphabet. Therefore, technically the two automata have synchronised on the event *e4*. As a final note shared events can be blocked in some states and not able to be synchronised on in all cases. From S0.S0 an *e1*-transition is possible since A_1 and A_2 both have outgoing *e1*-transitions from their corresponding states. However, an *e1*-transition is not possible from S1.S0. S0 of A_2 has an outgoing *e1*-transition, but S1 in A_1 does *not* have an outgoing *e1*-transition from S1. Such a situation means the shared event *e1* is *blocked* from occurring in S1.S0.

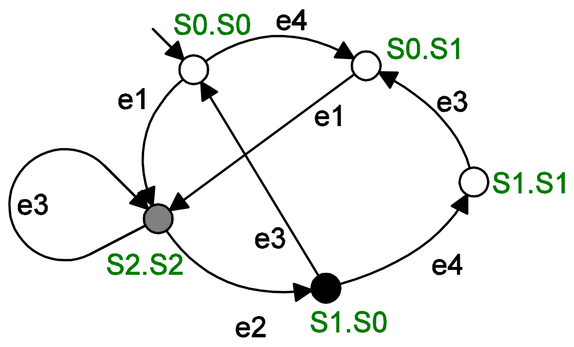

Figure 2.3: $A_1 \,||\, A_2$

2.3 Traces

A *trace* is a sequence of steps describing a particular behaviour of a system. A trace can be thought of as a path through an automaton which represents the system transitioning from one state to another (usually via other states). Every step of a trace has a particular state *x* as the source state and an event *e* which leads to a particular target state *y*. For an automaton to execute this step, its *x* state must have an outgoing transition, labelled *e*, which is incoming to *y*. *y* then becomes the source state for the next step and so on until the final state of the trace is reached. If all steps in a trace can be executed by an automaton in this way, the automaton accepts the trace. The automaton in Figure 2.1 accepts the following trace:

$$S0 \overset{e1}{\to} S2 \overset{e3}{\to} S2 \overset{e2}{\to} S1 \qquad (1)$$

The trace begins in the initial state of the automaton *S0*. The event *e1* is fired and the automaton is now in state *S2*. *e3* occurs in state *S2,* since *e3* labels a self-looped transition it does not affect the automaton's state. The final step of the trace is to move into state *S1* after *e2* happens. *S1* is termed the *end state* of this trace. Since the trace begins in an initial state of the automaton, the trace shows that *S0, S1* and *S2* are *reachable* states.

2.4 Standard and Generalised Nonblocking

It is usually desirable for systems to be free from *livelock* or *deadlock*. We can verify such a property for models with a single marking proposition by allocating particular states as terminal and then examining their reachability (Ramadge & Wonham, 1989).

This property is commonly known as *nonblocking* for a single finite-state automaton. However, the term nonblocking is used interchangeably with *nonconflicting* for a model containing multiple automata. Nonblocking translates to a set of automata, if from every reachable state all components *can* reach a terminal state in parallel.

A weakness of standard nonblocking is that its expressive power is quite limited for representing some problems. Consider a problem where we are only concerned with whether a terminal state can be reached if some precondition is satisfied. Standard nonblocking cannot model this, because it checks whether *every* reachable state can reach a terminal state. Therefore, in an effort to overcome this weakness, nonblocking has been generalised to handle multiple marking propositions using multi-coloured automata (Leduc & Malik, Generalised Nonblocking, 2008).

Definition Let $A = (\sum, \prod, X, \rightarrow, X^{\circ}, \Xi)$ with $\alpha, \omega \in \prod$, be a multi-coloured automaton. A is *generalised nonblocking* or (α, ω)-*nonblocking*, if for all states $x \in \Xi(\alpha)$ there exists a path from x to a state $y \in \Xi(\omega)$ (Leduc & Malik, Generalised Nonblocking, 2008). Otherwise, A is (α, ω)-*blocking*.

The similarities between nonblocking and its generalised form mean the same algorithm can verify both varieties. This is achieved by implementing a generalised nonblocking algorithm, and treating standard nonblocking as generalised. Standard nonblocking is treated as generalised nonblocking by considering all reachable states as α-marked. Thus, a *dummy* α-marking is added to the model's event alphabet and every state of every automaton in the model.

Example The automaton on the left in Figure 2.4 shows an example of an automaton which is (α, ω)-*nonblocking*. The only state we need to consider is *S2*, since it is the only α-marked state. The trace in (2) shows that *S2* can reach a terminal state (in this case the only terminal state) *S1*.

$$S2 \overset{e1}{\rightarrow} S3 \overset{e2}{\rightarrow} S1 \qquad (2)$$

The automaton on the right in Figure 2.4 shows an example of an automaton which is (α, ω)-*blocking*. Again, the only state we need to investigate is *S2*, since it is the only α-marked state. *S1* is the only terminal state and so a path must exist from *S2* to *S1* for the automaton to satisfy generalised nonblocking. From state *S2* the automaton is in a state of *livelock*, transitions are possible but from where the terminal state *S1* can never be reached. Thus, with the presence of livelock the system is (α, ω)-*blocking*.

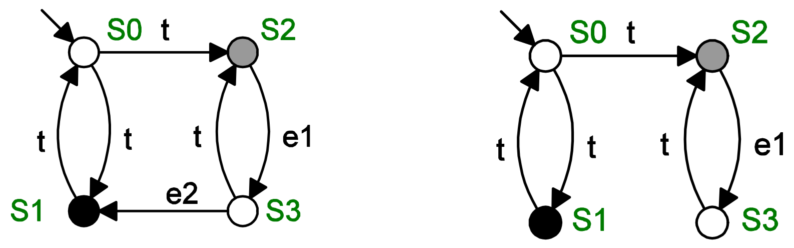

Figure 2.4: The automaton on the left is (α, ω)-*nonblocking*, the automaton on the right is (α, ω)-*blocking*.

2.5 Counterexamples

If an automaton is verified as (α, ω)-*blocking*, we want to be able to provide proof that the result is correct. A *counterexample* is a trace which proves a model is (α, ω)-*blocking*. Counterexamples also serve as a diagnostic which provides an explanation for why a system is (α, ω)-*blocking*. A trace *t* must adhere to the following conditions to be a valid counterexample for an automaton *A*:

1) The initial state of *t* must be an initial state of *A*.
2) From the specified initial state the automaton must be able to execute each step in the exact sequence presented by the trace.
3) The end state of *t* must be an α-marked state of *A*, from which no ω-marked state can be reached.

The trace in (3) is a valid counterexample for the automaton on the right in Figure 2.4. The trace correctly starts by specifying an initial state *S0* of the automaton. The *t*-transition specified can be executed by the automaton, satisfying the second requirement. Finally, *S2* is an α-marked state from where the only terminal state *S1* cannot be reached, because the system is in a state of livelock.

$$S0 \xrightarrow{t} S2 \tag{3}$$

3 Compositional Verification

Verifying generalised nonblocking of a system using a standard approach can be a straightforward process. Such a process involves explicitly building the synchronous product for the entire model and then examining whether for each α-marked state a reachable terminal state exists. Models of a reasonably substantial size are able to be verified with this approach if the state space is represented *symbolically* (Berard, et al., 2001).

The standard method has the major disadvantage that building the synchronous product of larger systems means processing a large number of states, which leads to the well-known *state space explosion problem* (Berard, et al., 2001). The state space explosion problem makes the standard method alone unable to verify generalised nonblocking of very large, real-world systems. Furthermore, for models which *are* small enough to be verified by a standard approach, the time taken to construct the synchronous product is exponentially proportional to the number of automata in the entire model. These two reasons justify why the standard approach is not viable for verifying larger systems and the need for a new approach.

A compositional approach attempts to increase the size of systems which can be verified, and to reduce the time taken to construct the synchronous product. This is done by reducing the size of a model prior to building the synchronous product. We can achieve this by composing several automata together and applying *conflict-preserving* abstraction rules to the composed automaton. Conflict-preserving is to retain all faults which existed in the original model, this ensures that reducing the size of a model will not affect the outcome of verifying generalised nonblocking in a final step. Composing several automata means a process must be in place for choosing which automata. This process identifies several automata as a *candidate* for composition.

Most abstraction rules rely on the existence of local events to make a significant abstraction. Thus, prior to abstraction an intermediate process called *hiding* is used. Hiding replaces all local events of an automaton by the silent event τ. The automata which were composed are then replaced by their *conflict equivalent* abstracted automaton in the model. Another choice is then made for the next automata to compose. Usually this process is discontinued when only one automaton remains. Finally, a standard approach is used to verify the simplified model. This compositional approach has already had considerable success in verifying standard nonblocking (Flordal & Malik, 2009) and we are extending these results to verifying generalised nonblocking.

Example The model shown in (1) could be verified compositionally as follows. We choose some (commonly two) individual components to compose, e.g. A_1 and A_2. Next we construct $A_1 \parallel A_2$ and abstract the resulting automaton. The simplified version $(A_1 \parallel A_2)'$ of $A_1 \parallel A_2$ then replaces A_1 and A_2 in the model, as shown by (2). Next the selection and abstraction process is repeated, usually until only one automaton remains.

$$A_1, A_2, \ldots, A_n \tag{1}$$

$$(A_1 \parallel A_2)', \ldots, A_n \tag{2}$$

An algorithm for the compositional approach discussed is outlined in Figure 3.1 using (1).

1. Choose some automata, e.g. A_1 and A_2, to compose.
2. Construct $A_1 \parallel A_2$.
3. Hide local events of $A_1 \parallel A_2$, to get $(A_1 \parallel A_2)'$.
4. Apply abstraction rules to get $(A_1 \parallel A_2)''$.
5. Replace A_1 and A_2 in the original model with the single automaton $(A_1 \parallel A_2)''$.
6. Repeat steps 1-5 as many times as preferred, or until only one automaton remains.
7. Verify generalised nonblocking of the simplified model using standard approaches.
8. If the result is false, expand the provided counterexample in the opposite order each abstraction occurred.

Figure 3.1: An algorithm for compositional verification of generalised nonblocking.

In the final step verification occurs for a simplified version of the model. Therefore, if the model is (α, ω)-*blocking*, a counterexample is produced which is accepted by the simplified model. Following abstraction of a model, it is very unlikely the same counter-example will be accepted by the original model. In order to provide a counterexample which will be applicable to the original system, we must be able to reverse all abstraction. Our implementation achieves this by storing a new *reversal step* after every kind of abstraction (including hiding, because this removes information from the model). We have different kinds of *reversal steps* to cater to each kind of abstraction; the reversal step stores the necessary information required to reverse its associated abstraction. To expand the counterexample given for the most simplified version of the model we must undo all abstraction steps in reverse order in which they occurred.

We intend to abstract very large candidates with our compositional verifier. Processing candidates of the size we expect can easily exceed memory allocations and require a long run time. Therefore we want to be able to control the size of candidates which can be processed. Our implementation can have *internal* state and internal transition *limits* set, as well as *final* limits to achieve this. *Internal limits* are used to the limit the number of transitions and states which can be explored during composition of a candidate and abstraction. If these limits are exceeded when processing a candidate it is marked as unsuccessful, and we will not allow this candidate to be selected again. We must also remove any reversal steps created for the candidate which was unsuccessful. This is because any simplification to the model is undone once a candidate becomes unsuccessful, and so we do not want to try and reverse this simplification if we need a counterexample. Finally, we attempt to find another suitable candidate to abstract. We stop selecting new candidates once only one automaton remains, or once there are no more possible candidates which have not been blacklisted. *Final limits* get passed on to the standard verifier used in the final step of our algorithm. If final limits are exceeded verification does not complete and reports an overflow occurring. Thus, our implementation is an extended version of the one in Figure 3.1.

The result of step 4 may be a *trivial automaton*, if $(A_1 \parallel A_2)'$ is abstracted completely. A trivial automaton is considered to be one with no transitions and a single state which contains both the α-marking and ω-marking. This automaton is (α, ω)-*nonblocking* and will not synchronise on any events in future, therefore we suppress it from the remaining computations of the algorithm.

To increase the understanding of our algorithm and for testing purposes it is important that the entire algorithm is deterministic. Thus, all our internal algorithms are designed in such a way that processing or selection of any kind is consistent for any one model. Our explanations of our algorithms for applying the various abstraction rules have been written with the assumption that all automata (and their states, transitions and events) are in some pre-defined ordering to guarantee this deterministic behaviour.

The following sections discuss the more complex steps of the algorithm in Figure 3.1 and their implementation in more detail.

3.1 Candidates

The first step of the algorithm is to choose which automata to compose. A selection process has been developed that attempts to choose the best possible automata to compose, given a model. Several automata which are to be considered for composition are collectively known as a *candidate*. The best candidate is that which is predicted to have potential for the most simplification and/or the smallest synchronous product state space. The strategy for doing this is to compose the candidate that will possess as many local events as possible. Later, when the abstraction rules are discussed, it will become clear why a high proportion of local events is thought to yield the most simplification. On the other hand, choosing the candidate with the most shared events will lead to a smaller synchronous product, since synchronisation can occur for a larger number of events. Selecting candidates for composition has been developed as a two-step process. There are many possibilities for heuristics to implement which suit both steps. We implemented many different heuristics and variations of them for this project. We carried out experiments to evaluate which heuristics and what combinations of them perform better. Section 4.3.1 discusses these experiments and results.

The first step of candidate selection is to compute some automata to pair/group together as candidates for composition. Step one is used to avoid considering all possible combinations in step two. The second step attempts to identify the best candidate. Our implementation allows the user to configure which heuristic to use at both steps. In step two, if the chosen heuristic fails to identify only one candidate then another heuristic is used to choose between the top candidate choices given by the chosen heuristic. This process is repeated until all heuristics are used and then our implementation falls back on a default heuristic. Our default heuristic is to order the candidates automata names lexicographically and choose the first one, as this will guarantee a single candidate is selected in a deterministic manner.

Once a candidate is chosen, the synchronous product is computed and a set of abstraction rules are applied.

3.2 Hiding

Hiding is a simple form of abstraction which replaces the local events of an automaton with the silent event τ. In general, hiding will introduce nondeterminism. This happens when a state has multiple outgoing transitions which are labelled with local events. Any transition labelled τ does not require any event to occur to transition from the source state to the target state. To be conflict-preserving, many of the abstraction rules perform abstractions with respect to τ–transitions. This is why hiding must be used prior to the abstraction rules we implemented.

Example Figure 3.2 shows the result of hiding if the local events for *A* are *e2* and *e3*. The outgoing transitions from S1 illustrate how nondeterminism can be introduced. After hiding when the system is in state S1, if the event *e4* is fired, the system can move to state S3 via S2 since the τ-transition does not require an event to occur to move between these states. In the same way, if *e5* occurred while in S1 the system transitions to S0 via S3, this time using the other τ-transition.

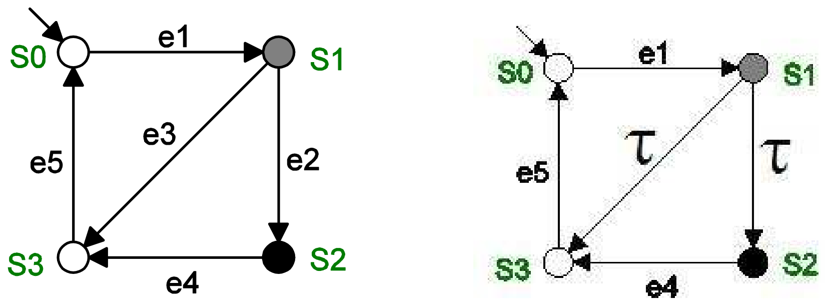

Figure 3.2: Automaton *A* on the left, and on the right the result of hiding.

Using our iterative compositional approach, hiding will usually be performed many times. This raises the need to use unique τ-event for hiding for each iteration. Otherwise when synchronously composing two previously simplified automaton (A_1 || A_2) and (A_3 || A_4), both would share the same τ-event which would erroneously be considered a shared event and would incorrectly allow synchronised execution of this event in the new composed automaton ((A_1 || A_2) || (A_3 || A_4)).

Hiding abstracts events out of an automaton, resulting in a counterexample which will specify τ-events that did not exist in the original automaton. Thus, we need to be able to recover the events which were replaced on transitions, should a counterexample be necessary. To achieve this, the original automaton's local events must be stored as well as the unique τ-event used for this particular iteration.

Expanding a counterexample which was given for the simplified automaton to an equivalent one for the original automaton requires replacing any τ-events associated with the trace's steps. This can be done by iterating over the following two cases for each step of the trace, starting from the initial state specified by the counterexample. If the step's event is not τ the step is already equivalent for the original automaton. Alternatively, if the step's event is τ, using the target state of the previous step (or initial state for the first step) as the source state and the target state specified by this step, we search the original automaton for a transition which connects these two states and is labelled with a local event. We then replace τ with the local event found.

3.3 Abstraction Rules

Seven conflict-preserving abstraction rules are presented in (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009) and (Leduc & Malik, A compositional approach for verifying generalised nonblocking, 2009). These rules were designed to be computationally feasible and able to achieve a fair reduction of the state space. While they were specifically developed for generalised nonblocking, they are also applicable to standard nonblocking. Proof of these rules correctness and complexity is available in (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009). We implemented and experimented with all seven of these rules.

Alongside these, we used an eighth rule, τ-loop removal, which is a specific case of the first rule.

In this chapter we outline the set of rules we implemented for abstracting an automaton into a (usually) simplified version, which is (α, ω)-nonblocking if and only if the original automaton is. Each rule's description describes the rule in the way it is presented in (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009) and gives a simple example of the rule's application. As well as this we describe two algorithms; the first algorithm is what we implemented for applying the rule, the second algorithm we implemented for expanding a counterexample given after the rule's application. The definition for several of these rules states how to simplify a single state, as opposed to an entire automaton. For those rules, the algorithm we implemented will apply the rule whenever possible for a given automaton, not only to one state.

3.3.1 Observation Equivalence

Observation equivalence or weak bisimulation is known as one of the most robust equivalences of nondeterministic automata (Milner, 1980). Even applying the observation equivalence abstraction rule alone (or alongside other abstraction rules) can achieve a considerable reduction in the number of states of an automaton. Abstraction is completed with respect to an equivalence relation. The theory is to categorise particular states as equivalent and group them together to be merged into a single state. Observation equivalence regards two states as equivalent if they have precisely the same formation of future nondeterministic behaviour. Future behaviour refers to the possible paths for execution after entering some state. Observation equivalence is described formally below.

Definition Let $A_1 = (\sum, \prod, X_1, \rightarrow_1, X_1^\circ, \Xi_1)$ and $A_2 = (\sum, \prod, X_2, \rightarrow_2, X_2^\circ, \Xi_2)$ be two multi-coloured automata. Let $s \overset{e}{\Rightarrow} t$ denote a path, where e is a string of events, for which there is an arbitrary number of τ events shuffled with the events of e. A relation $\approx\ \subseteq X_1 \times X_2$ is a weak bisimulation between A_1 and A_2 if, for all states $x_1 \in X_1$ and $x_2 \in X_2$ such that $x_1 \approx x_2$,

- if $x_1 \overset{e}{\Rightarrow}_1 y_1$ for some $e \in \sum^*$, then there exists $y_2 \in X_2$ such that $y_1 \approx y_2$ and $x_2 \overset{e}{\Rightarrow}_2 y_2$;
- if $x_2 \overset{e}{\Rightarrow}_2 y_2$ for some $e \in \sum^*$, then there exists $y_1 \in X_1$ such that $y_1 \approx y_2$ and $x_1 \overset{e}{\Rightarrow}_1 y_1$;
- if $x_1 \in \Xi_1(\pi)$ for some $\pi \in \prod$, then $x_2 \overset{\epsilon}{\Rightarrow}_2 \Xi_2(\pi)$;
- if $x_2 \in \Xi_2(\pi)$ for some $\pi \in \prod$, then $x_1 \overset{\epsilon}{\Rightarrow}_1 \Xi_1(\pi)$;

A_1 and A_2 are observation equivalent, $A_1 \approx A_2$, if there exists a weak bisimulation $\approx$ between A_1 and A_2 such that, for each initial state $x_1^\circ \in X_1^\circ$ there exists $x_2^\circ \in X_2$ such that X_2°

$\overset{\epsilon}{\Rightarrow}_2\, x_2^\circ$ and $x_1^\circ \approx x_2^\circ$, and vice versa (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009).

Rule If two automata A_1 and A_2 are observation equivalent then A_1 can be replaced by A_2 (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009).

The definition for this rule may not make it immediately obvious how its application can be so powerful in terms of simplification. Particularly since we have said our compositional approach is to compose several automata into one automaton which will be abstracted. A single automaton A_1 has its states analysed with respect to observation equivalence. Multiple states which are found to be equivalent can be merged into one state producing a simplified automaton A_2. A_2 will be observation equivalent to A_1 and therefore can replace A_1 in the compositional process. Alongside analysing states, transitions are also explored. This is useful because hiding introduces redundant transitions which can be disconnected (Eloranta, 1991).

Example Figure 3.3 shows one example of what observation equivalence can achieve. s_1 is treated as an initial state since it has an incoming τ-transition from the initial state s_0. The explicit (s_0) and implicit (s_1) initial states both have the same future behaviour (a single outgoing e_1-transition), thus they are observation equivalent states. Therefore, for automaton A_1, s_0 can be merged with s_1 and the τ-transition between s_0 and s_1 in A_1 is abstracted out of the automaton. After this step, two e_1-transitions would remain from the initial state s_0 to the only remaining state s_2. One of these e_1-transitions can be removed since it is redundant, leaving us with the simplified automaton A_2.

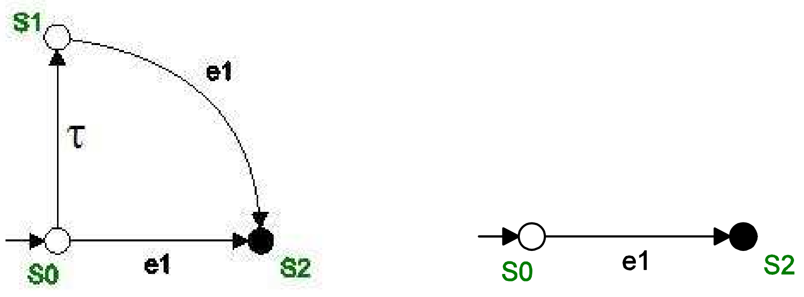

Figure 3.3: Automaton A_2 (right) shows automaton A_1 (left) after applying the observation equivalence rule.

Algorithm

The properties of observation equivalence offer efficient, intelligent algorithms making its application less expensive than might be expected. For this project an existing implementation, which was developed following the ideas in (Fernandez, 1990) and (Eloranta, 1991) was used for applying the observation equivalence rule.

Counterexample Expansion

A trace provided after application of the observation equivalence rule can potentially have a lot of information missing, information which is necessary for the counterexample to be accepted by the original automaton. Therefore, this trace expansion implementation is quite difficult and time-consuming to achieve correctly. The expansion procedure can be thought of as three separate sub-steps. Each section of the expansion involves breadth-first searching and adding any τ-transitions traversed to the expanded counterexample.

Let $A = (\sum, \prod, X_A, \rightarrow_A, X_A^\circ, \Xi_A)$ and $B = (\sum, \prod, X_B, \rightarrow_B, X_B^\circ, \Xi_2)$ be two multi-coloured automata. A is abstracted with respect to observation equivalence, resulting in automaton B, such that $B \approx A$. The simple counterexample shown by (1) is given as an example for the abstracted automaton, B. Since $A \approx B$, we know that for each initial state $x_B^\circ \in X_B^\circ$ there exists $x_A^\circ \in X_A$ such that $X_A^\circ \overset{\epsilon}{\Rightarrow}_A x_A^\circ$ and $x_A^\circ \approx x_B^\circ$. Therefore, the first step of an observation equivalence trace expansion is to conduct a breadth-first search from each initial state $i_A^\circ \in X_A^\circ$ to find $x_A^\circ \in X_A$ where $x_A^\circ \approx x_B$. If necessary, extra τ-steps are added to the expanded counterexample to represent the trace $X_A^\circ \overset{\epsilon}{\Rightarrow}_A x_A^\circ$.

$$x_B^\circ \overset{\sigma 1}{\rightarrow} y_B \overset{\sigma 2}{\rightarrow} z_B \tag{1}$$

Following the observation equivalence definition, if $x_B \overset{e}{\Rightarrow}_B y_B$ for some string of events $e \in \sum^*$, then there exists $y_A \in X_A$ such that $y_A \approx y_B$ and $x_A \overset{e}{\Rightarrow}_A y_A$. Hence, the second step for trace expansion is to conduct a breadth-first search from $x_A^\circ \in X_A$ to find $y_A \in X_A$ such that $x_A^\circ \overset{\sigma 1}{\Rightarrow}_A y_A$ and $y_A \approx y_B$. If necessary, extra τ-steps are added to the expanded counterexample to represent the trace $x_A \overset{\sigma 1}{\Rightarrow}_A y_A$. This second step is repeated for every remaining step of B's counterexample (only once more for our trivial example).

Finally, a counterexample representing (α, ω)-*blocking* must end in an α-marked state. Therefore, B's counterexample end state z_B will be α-marked and the expanded counterexample must also be α-marked. After step two is completed a state z_A is found such that $z_A \approx z_B$. However, the definition says if $z_B \in \Xi_B(\alpha)$ for $\alpha \in \prod$, then $z_A \overset{\epsilon}{\Rightarrow}_A \Xi_A(\alpha)$. Thus, z_A may not be α-marked and one last breadth-first search is required from z_A to add any necessary τ-steps which are included in $z_A \overset{\epsilon}{\Rightarrow}_A \Xi_A(\alpha)$, to the expanded counterexample.

3.3.2 τ-Loop Removal

A τ-*loop* is a series of τ-transitions which are *strongly connected.* A set of states are *strongly connected* if there is a path from each state in the set to every other state. Therefore, a τ-loop is a special case of observation equivalence.

Rule A τ-loop can be replaced by a single state, if all types of marking propositions encountered in the τ-loop are added to the remaining state.

This rule is implicitly applied by an observation equivalence algorithm. However, we chose to apply it using a separate algorithm. The reason for this being that an even more efficient algorithm exists than the one used for computing observation equivalence, which can identify this specific case of observation equivalence. Also, one of the following rules' algorithms depends on an automata having no τ-loops, thus sometimes we may want to remove τ-loops but without computing observation equivalence.

Example Figure 3.4 shows an example of a τ-loop and how the rule can be used. A_1 has the trace in (1) as a τ-loop. We can remove all transitions and all states (except one) which are part of this τ-loop. A_2 shows that *S1* is the state maintained. *S3* of A_1 has an ω-marking, *S1* and *S4* of A_1 have α-markings. Thus, *S1* of A_2 must be marked with an ω-marking and an α-marking to be (α, ω)-nonblocking equivalent to A_1. *S1* of A_2 depicts a state with two marking propositions.

$$S1 \xrightarrow{\tau} S3 \xrightarrow{\tau} S4 \xrightarrow{\tau} S5 \tag{1}$$

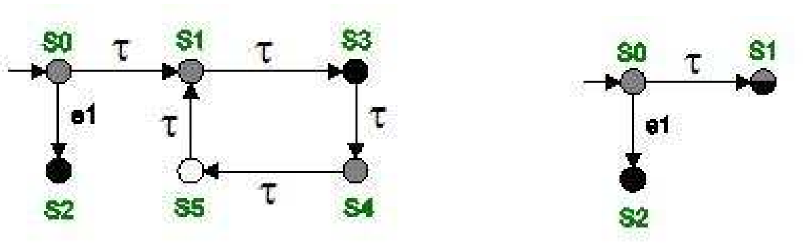

Figure 3.4: Automaton A_2 (right) shows automaton A_1 (left) after applying the τ-loop removal rule.

Algorithm

An existing implementation of Tarjan's algorithm for finding the strongly connected components in a directed graph (Tarjan, 1972) was used for applying this rule.

Counterexample Expansion

Because removing τ-loops is a special case of observation equivalence, the same algorithm used for expanding a counterexample given after applying the observation equivalence rule can be used for expanding a counterexample here (please refer to section 3.3.1 to see reasoning for the implementation of this algorithm).

3.3.3 Removal of α-markings

While observation equivalence achieves a great reduction of the state space, there are automata which are not observation equivalent but are (α, ω)-*nonblocking* equivalent. Therefore, further rules are desired to apply to these automata. The remaining rules are applied directly to the states and transitions of an automaton.

Rule If an automaton contains two different states *x* and *y* both marked α, such that $x \xrightarrow{\tau} y$, then the α-marking can be removed from state *x* (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009) .

While the removal of markings does not directly reduce the size of an automaton, it can enable other abstraction rules to simplify further. Also verification is expected to be quicker and easier with fewer states α-marked given that for verifying generalised non-blocking we are only interested in visiting paths from α-marked states. Using this rule on a standard nonblocking model being treated as generalised nonblocking can potentially remove many markings, since all states are marked α initially. In turn, this may make other rules become applicable, rules which were not capable of simplification prior to the removal of α-markings.

Example Figure 3.5 shows an example of applying the removal of α-markings rule. *S0* and *S1* are the only α-marked states of A_1, thus these are the only two we can simplify. *S0* can have its α-marking removed since $S0 \xrightarrow{\tau} S1$ and *S1* is also α-marked. *S1* cannot have its α-marking removed since it has no outgoing τ-transition and its only target state is not α-marked.

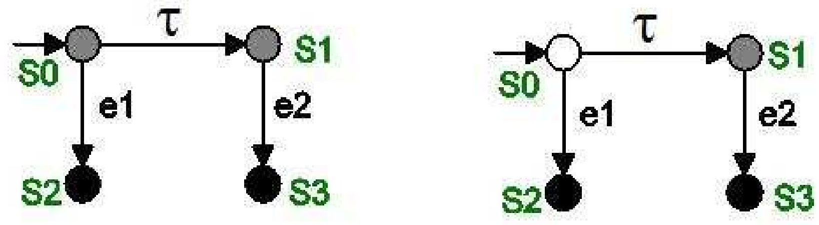

Figure 3.5: Automaton A_2 (right) shows automaton A_1 (left) after applying the removal of α-markings rule.

Algorithm

Figure 3.7 outlines the algorithm we implemented for applying this rule multiple times to an automaton. The algorithm will remove all α-markings which qualify for removal from an automaton. A backwards search is used from every state in the automaton. Prior to this we perform a preliminary check. The rule states that an α-marking can only be removed, if there is at least one τ-transition between the two states concerned. Thus, if the automaton's event alphabet does not contain τ, we do not apply the rule.

There are two main data structures used to keep track of states visited/not visited during the search. A stack *U,* is used during the traversal from each state to record states which have not been visited yet. A Hash Set *R,* is used to keep track of states which were reached via a traversal of the current state being processed.

A main loop iterates over every state *s* in the automaton *A*. Within that loop if *s* is α-marked there is a chance of removal, therefore *s* is pushed on to *U* so that a backwards search of *s'*s τ-predecessors can follow. The first state *n* is removed from *U*. Removal of an α-marking can only occur from predecessor states reachable by at least one transition labelled τ, making the next step to get all immediate τ-predecessors of *n*. The predecessors are then iterated over to determine if they need to be checked for τ-predecessors deeper and whether an α-marking can be removed from any of them. However, before adding a predecessor to *U* to continue searching from or removing an α-marking, we must check that the predecessor state is not the state itself.

A state can be its own predecessor in two cases. The first case is that the state has a self-loop, meaning the source and target are not different. Therefore, the α-marking must remain. Or in the second case, the state is part of a τ-loop such as *S1* of A_1 in Figure 3.6. In the case of a τ-loop, all α-markings of a loop can be removed, except for the state which the search began from, *S1* in the example. Actually, any state in a complete τ-loop could remain α-marked; A_1 and A_2 would still be (α, ω)-*nonblocking* equivalent. We have chosen to always retain the α-marking on the state from where the backwards search begins, this ensures deterministic behaviour of the algorithm. Once confirmed that the predecessor is not *n* nor *s* there are two things to verify. If the predecessor can be added to *R* then it is added and also pushed on to *U*.

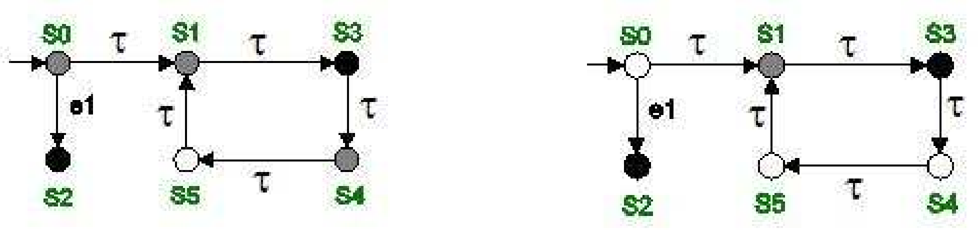

Figure 3.6: Automaton A_2 (right) shows automaton A_1 (left) after applying the removal of α-markings rule.

The purpose of adding a predecessor to *U* is to see if *s* has τ-predecessors more than one transition deep and so further searching is required from the predecessor. Furthermore, adding to *R* ensures that a predecessor is not repeatedly pushed onto *U* and searched multiple times indefinitely or unnecessarily. Finally, we check if the predecessor is α-marked, if it is then the α-marking is removed. After searching a state and all its relevant predecessors *R* must be cleared so that the states which still need to be searched can check all predecessors to be considered.

Let *A* be the original automaton before abstraction, α be the α-marking in *A*'s alphabet, τ be the silent non-event used for hiding in *A, U* be a stack containing unvisited states and *R* be a HashSet containing states reached during the search from a state.

```
if (τ is not in A's alphabet) {
    return;
}
if (α is not in A's alphabet) {
    explicitly mark every state in A with a 'dummy' α–marking;
}
for each (State s of A's states) {
    if (s is α-marked) {
        push s onto U;
        while (U's size > 0) {
            pop state n off U;
            for each (τ-predecessor state p of n) {
                if ((p != s) && (p != n)) {
                    if (p is not in R) {
                        add p to R;
                        push p onto U;
                    }
                    if (p is α–marked) {
                        remove α–marking from p;
                    }
                }
            }
        }
    }
    clear R;
}
```

Figure 3.7: Removal of α–markings Algorithm

Counterexample Expansion

A counterexample produced by a conflict checker after removal of α–markings has been applied will not have any information missing since no states or transitions are removed by this rule. However, we must be able to identify the objects used for states in the automaton before the rule was applied which correspond to the states given by the trace. Events do not need to be replaced because events are never modified by rules. Figure 3.8 outlines an algorithm for expanding a counterexample given for an automaton simplified using the removal of α–markings rule into a counterexample accepted by the original version of the automaton.

Let A_1 be the original automaton before abstraction, A_2 be the simplified automaton after abstraction, C be the counterexample for the simplified automaton, T be the list of trace steps in the counterexample, L_1 be a sequentially ordered list of steps applicable to A_1, and L_2 be the list of automata the counterexample applies to.

```
1.    for each (step $s$ in $T$) {
2.        remove $A_2$ from $s$'s state map $M$;
3.        add $A_1$ to $M$, mapped to $A_1$'s equivalent state for the $A_2$ state removed;
4.        create a new Step $s_1$ using $s$'s event and $M$;
5.        add $s_1$ to $L_1$;
6.    }
7.    create a new counterexample using $L_1$ and $L_2$;
```

Figure 3.8: Removal of Markings Trace Expansion Algorithm.

3.3.4 Removal of ω-markings

We can remove ω-markings when an ω-marked state is not reachable from any α-marked state.

Rule If a state x is not reachable from any state marked α, then an ω-marking can be removed from (or added to) state x (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009).

Just as with removing α-markings, removing ω-markings will not directly reduce an automaton's size and is instead used in the hope of increasing the abstraction potential of other rules. The removal of ω-markings can sometimes increase the number of non-coreachable states, making the next rule we discuss more powerful when applied after removing ω-markings. Generalised nonblocking verification means we are only concerned with visiting paths from α-marked states. Therefore, as the rule suggests we can add an ω-marking to any state x which is not reachable from an α-marked state, without affecting the generalised nonblocking property of the model. However, adding these markings is a misuse of effort since it will not make the verification more efficient or allow our other rules to abstract any differently than they would without the marking. Hence, we are only concerned with using this rule to remove ω-markings.

Example Figure 3.9 shows an example of applying the removal of ω-markings rule. *S2* and *S3* are the only ω-marked states of A_1, thus these are the only two which have the potential to be simplified. *S2* can have its ω-marking removed since it is not reachable from any α-marked state. *S3* cannot have its α-marking removed since it is reachable from the α-marked state *S1*.

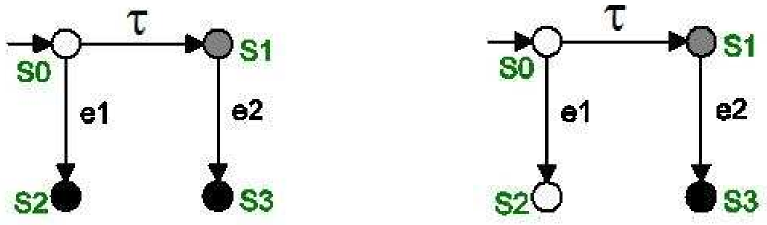

Figure 3.9: Automaton A_2 (right) shows automaton A_1 (left) after applying the removal of ω-markings rule.

Algorithm

Figure 3.10 outlines the algorithm we implemented for applying this rule. The idea behind the algorithm for removal of ω-markings is to find all states which can be reached from an α-marked state, and then remove ω-markings from any states which were not found as reachable. This is achieved by conducting a forwards search from every state in the automaton. However, prior to beginning the search there is a check which can be done to avoid unnecessary application of the rule. We check if there is an α-marking in the automaton's alphabet. When verifying generalised nonblocking an automaton without an α-marking included in its event alphabet means that all states are regarded as being implicitly α-marked. For this reason, the rule will not be able to remove any ω-markings since all states can certainly be reached from an α-marked state (the state itself).

We use two main data structures throughout the search. A stack *U,* is used during the traversal from each state to record states which have not been visited yet. And a Hash Set *R,* is used to record which states were reached from some α-marked state being searched.

The forward search iterates over every state belonging to the automaton to be abstracted. If the current state is α-marked we are interested in finding all states which can be reached from it. However, to avoid repeating a traversal we check that the state has not already been traversed from a previous α-marked state. From the α-marked states we visit their immediate successor states and record each successor as reachable and "to be visited". We then perform the same search for immediate successors (the states which are now marked as to be visited) from each immediate successor of the α-marked state. This deep searching is necessary because only searching immediate successors from each state would mean many states which are reachable from α-marked states over several transitions will not become marked as reachable, as the search would only cover immediate successors. This forward search completes the first phase of the algorithm; finding all states reachable from α-marked states.

The next step of the algorithm is to check for every state in the automaton if it was found as reachable. If the state was not reachable and is ω-marked we remove the ω-marking from it.

Let *A* be the automaton to be abstracted, α be the α-marking in *A*'s alphabet, ω be the ω-marking in *A*'s alphabet, *U* be a stack containing unvisited states and *R* be a HashSet containing states reached during the search from a state.

```
1.  if ($A$'s alphabet does not contain $\alpha$) {
2.      return;
3.  }
4.  for each (state $s$ in $A$'s states) {
5.      if (($s$ is marked $\alpha$) & ($R$ does not contain $s$)) {
6.          push $s$ on to $U$;
7.          add $s$ to $R$;
8.          while ($U$ is not empty) {
9.              pop state $n$ off $U$;
10.             for each of ($n$'s immediate successor states $t$) {
11.                 if ($R$ does not contain $t$) {
12.                     push $t$ on $U$;
13.                     add $t$ to $R$;
14.                 }
15.             }
16.         }
17.     }
18. }
19. for each (state $s$ in $A$'s states) {
20.     if (($s$ is marked $\omega$) & ($R$ does not contain $s$)) {
21.         remove $\omega$-marking from $s$;
22.     }
23. }
```

Figure 3.10: Removal of ω-markings Algorithm.

Counterexample Expansion

Counterexample expansion after applying removal of ω-marking uses the same algorithm that removal of α-marking uses, please refer to section 3.3.3 to see the algorithm and an explanation. Using the same algorithm works since the only modification to be made is the same for both rules, irrespective of whether the abstraction removed an α marking or removed an ω-marking the expansion is the same.

3.3.5 Removal of Non-coreachable States

This is the first rule we discuss which actually removes states from an automaton, and so is directly reducing the automaton's size. Removal of Non-coreachable states will usually have greater potential for removal of states when applied after Removal of ω-markings, since removing ω-markings often increases the number of non-coreachable states. (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009).

Rule States that are not α/ω-coreachable, i.e., from which neither a state marked α nor a state marked ω can be reached, can be removed (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009).

Example Figure 3.11 shows an example of applying removal of non-coreachable states. State *S2* in A_1 is neither α-coreachable nor ω-coreachable. Therefore, *S2* and its incoming transitions can be removed, as shown by A_2.

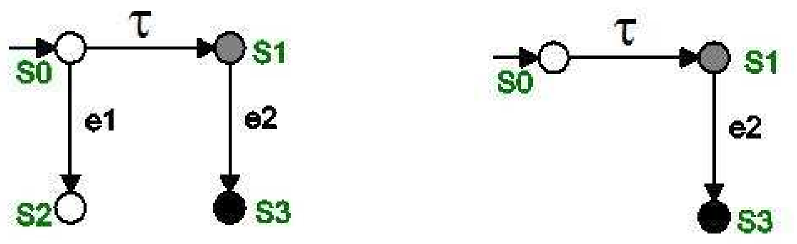

Figure 3.11: Automaton A_2 (right) shows automaton A_1 (left) after applying the removal of non-coreachable states rule.

Algorithm

The removal of non-coreachable states' algorithm is similar to that for removal of ω-markings. A search is performed from particular states of the automaton, recording states which are traversed as co-reachable and then performing the actual modification after the search.

Before traversing the automaton we check if its alphabet excludes either the ω-marking or the α-marking. As discussed earlier, if a marking is not explicitly part of the event alphabet then all states are implicitly marked by it. Therefore, all states can reach a state with one of the required markings and no simplification is possible, in this case we return immediately.

We search backwards from any state which has an ω-marking or α-marking to find the states which are able to reach them. For a state to be worthy of searching it must not have been previously marked as α/ω-coreachable. A state which was already recorded as α/ω-coreachable has had all its predecessors searched previously, and so for efficiency we want to avoid repeating a traversal when unnecessary. Once a state passes all conditions for being worthy of searching it is pushed onto the stack of states to be visited, this indicates a search needs to be performed for this state. The state is also added to the list of α/ω-coreachable states so that it will not be traversed again in future and not incorrectly removed.

A state is taken from the unvisited states stack and has the following conditions checked for all of its immediate predecessor states. If the predecessor has not previously been processed, then we add this predecessor to the α/ω-coreachable states list and the unvisited states stack. The process of removing a state from the stack is then repeated for all predecessors of the state which was initially being searched before continuing with searching from the next state with one of the two markings.

After a backwards search is completed from all significant states we can check for every state of the automaton if it was found to be α/ω-coreachable reachable or not. If a state was not α/ω-coreachable that state is removed from the automaton.

Let *A* be the automaton to be abstracted, α be the α-marking in *A*'s alphabet, ω be the ω-marking in *A*'s alphabet, *U* be a stack containing unvisited states and *R* be a HashSet containing states reached backwards from α/ω-marked states.

```
if (A's alphabet does not contain α || A's alphabet does not contain ω) {
    return;
}
for each (State s in A's states) {
    if (((s is marked α) || (s is marked ω))  && (R does not contain s)) {
        push s on to U;
        add s to R;
        while (U is not empty) {
            pop state n off U;
            for each of (n's immediate predecessor states p) {
                if (R does not contain p) {
                    push t on U;
                    add t to R;
                }
            }
        }
    }
}
for each (state s in A's states) {
    if (R does not contain s) {
        remove state s;
    }
}
```

Figure 3.12: Removal of Non-coreachable States algorithm.

Counterexample Expansion

Counterexample expansion after applying removal of non-coreachable states uses the same algorithm as removal of α–markings and removal of ω-markings, even though this rule removes states and the other two rules remove markings; please refer to section 3.3.3 to see the algorithm and an explanation of it. We are able to use the same algorithm because the states which this rule removes will not appear in the counterexample, thus no extra information for the trace is missing.

3.3.6 Determinisation of Non-α States

In generalised nonblocking there are two different kinds of states. α-marked states carry nonblocking requirements, and their precise nondeterministic future may be relevant. In contrast, non-α states do not carry nonblocking requirements, making only the language associated with these states important (Leduc & Malik, A compositional approach for verifying generalised nonblocking, 2009)

Rule Two non-α states that are reachable by exactly the same strings from initial states and from each state marked α, can be merged into a single state (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009).

For two states to be reachable by exactly the same strings from some state *s*, both states must have the same number of transitions leading to them from *s*, and the transitions must be labelled with identical events in sequence.

Example Figure 3.13 demonstrates the result of applying the Determinisation of Non-α States rule to automaton A_1. A_1 has states s_1 and s_2 which are not α-marked and can only be reached from the sole initial (and in this case the only α-marked) state s_0 by the string e_1. Therefore, s_1 and s_2 are merged into the single state s_3 in A_2. No other states can be merged as they are reached by different strings from the initial/α-marked state.

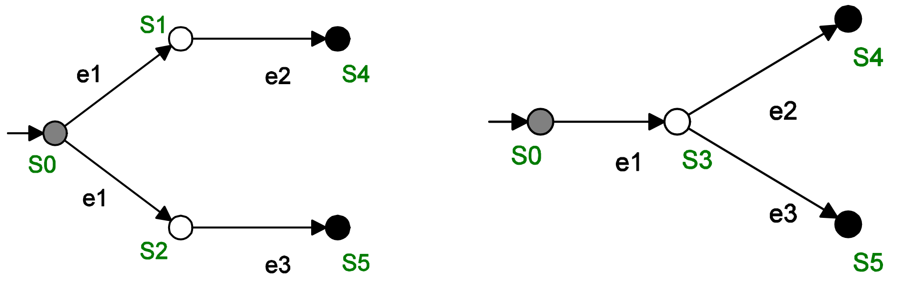

Figure 3.13: Automaton A_2 (right) shows automaton A_1 (left) after applying the Determinisation of Non-α States rule.

Determinisation of Non-α States follows the same concept as observation equivalence, but employs a reverse weak bisimulation as opposed to a weak bisimulation. A reverse weak bisimulation regards two states as equivalent if they can be reached by executing traces containing exactly the same strings from all initial states (Wen, Wang, & Qi, 2004). Determinisation of Non-α States handles simplification restricted by preceding traces from initial states, whereas observation equivalence is constrained by the traces succeeding a state. Therefore, Determinisation of Non-α States can be computed in the same way as observation equivalence using the algorithm in (Fernandez, 1990), under additional constraints relevant to this rule which are discussed below.

Algorithm

An existing implementation of the algorithm in (Fernandez, 1990) was used for this project to compute determinisation of non-α states, this is the same implementation as we used for computing observation equivalence. By default, the implementation computes observation equivalence. The states which can be merged are more specific for determinisation of non-α states than for observation equivalence. Hence, extra work is required before applying the algorithm so that it abides by the additional constraints of Determinisation of Non-α States. The implementation of Fernandez's algorithm which we used can have an *initial partition* specified. A *partition* contains a set of *equivalence classes*. An *equivalence class* is a set of states which are considered equivalent. The algorithm can compute other equivalence relations by ensuring the initial partition is satisfied in the result.

Before using the Fernandez algorithm, we reverse the direction of every transition in the automaton to be abstracted, since Determinisation of Non-α States is a *reverse* weak bisimulation. Next we create an initial partition, which consists of equivalence classes specific to this rule. For determinisation of non-α states the number of equivalence classes varies from automaton to automaton. The rule specifies that only non-α states can be merged. Such a specification is essential because of the nonblocking requirements α-marked states carry. Thus a separate equivalence class is created for every state marked α. If any α-marked state was a member of an equivalence class containing any other state, the algorithm would incorrectly consider these states for merging. One further equivalence class is created containing all non-α states, these states are the only ones which have the potential to safely be merged. We then refine this partition with respect to initial states. This refinement splits the equivalence class which contains all non-α states into an equivalence class of initial states, and a separate equivalence class of non-initial states. This prevents initial states from being merged with non-initial states. This refinement is necessary as a requirement of reverse weak bisimulation. All equivalence classes containing a single α-marked state are maintained and added to the refined initial partition. Finally, the result can be computed using the same Fernandez algorithm as for observation equivalence.

Counterexample Expansion

To expand a counterexample given after applying determinisation of non-α states to an automaton, we can use the reasoning given for expansion after applying observation equivalence (please refer to section 3.3.1). However, the third step is not necessary. This is because determinisation of non-α states does not merge α-marked states, and therefore we know the counterexample provided after extraction will be an α -marked state. The reasoning behind these two expansions is closely related since both are equivalence relations. Both expansions require the same kind of activity, using breadth-first searching to add any necessary τ-steps to the counterexample. Thus, this expansion algorithm can be very similar to that of observation equivalence counterexample expansion. The same reasoning is used, except that the expansion is done in reverse and we do not need to search for an α-marked state as the end state. For all other rules we expand the counterexample from the initial state, but in this case we expand from the end state of the counterexample. This is necessary since we reversed the direction of all transitions in the automaton before abstraction.

3.3.7 Removal of τ-Transitions Leading to Non-α States

τ–transitions introduced by hiding allow great potential for abstraction. Non-α determinisation can merge the source and target state of silent transitions which connect two states that are both not α-marked. Alternatively, if both states are α-marked, the Removal of α-Markings rule can be applied and will remove the α-marking of the source state. Removal of τ-Transitions Leading to Non-α States, and the similar rule presented in the next section, are able to provide simplification where at most one of the two states linked by a silent transition is α-marked (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009).

Rule A transition $x \xrightarrow{\tau} y$, with y not α-marked can be removed if all transitions originating from state y are copied to state x (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009).

This rule once again shows that most rules are not significantly powerful independently, but instead complement one another. Using this rule to remove τ-transitions does not guarantee a reduction in state space or complexity of the model, in fact for this rule it is unlikely this will happen. Removing a τ-transition results in all the target state's transitions being copied to the source state, this ensures the simplified automaton is (α, ω)-nonblocking equivalent to the original version. However, by doing this we are increasing the number of transitions in the model. The main benefit of this rule is the chance that removing a τ-transition may make the target state unreachable, perhaps after several applications. Secondly, the modified version of the model will have transitions more regularly structured which may increase simplification potential of other rules (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009).

Example Figure 3.14 shows an example of the application of the Removal of τ-Transitions Leading to Non-α States rule. A_1's transition $S0 \xrightarrow{\tau} S1$ leads to a non-α state, therefore it can be removed after the e_2-transition (the only transition originating from the target state *S1*) is copied to the source state *S0*. Consequently *S1* becomes unreachable and can be removed as shown.

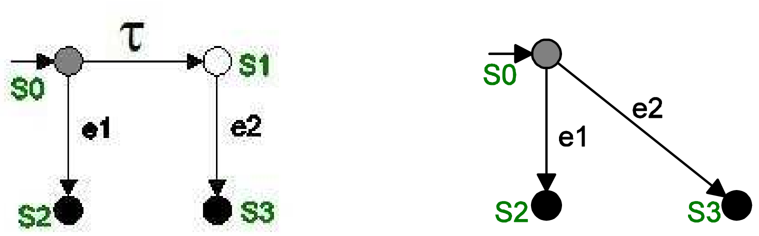

Figure 3.14: Automaton A_2 (right) shows automaton A_1 (left) after applying the Removal of τ-Transitions Leading to Non-α States rule.

Algorithm

This algorithm assumes that before using it, any τ-loops have been removed (please refer to section 3.3.2 for the τ-loop removal algorithm). If τ-loops existed in the automaton this particular rule may repeatedly remove a τ-transition and then redirect another τ-transition to the state where the τ-transition was just removed, resulting in execution being stuck in an infinite loop.

The first step of the algorithm is to check that the automaton's event alphabet contains an α-marking. Without explicit inclusion of an α-marking no modification is possible, all states will be implicitly α-marked and the rule stipulates that τ-transitions can only be removed if the target state is not α-marked. Secondly, the automaton's alphabet must contain a τ event to proceed with the algorithm; this avoids needlessly performing a search of an automaton with no silent transitions.

A queue is used to keep track of the states whose transitions we still want to analyse. For every state we only search one transition deep as this rule does not depend on any type of reachability of states, unlike several other rules. Therefore, we are able to remove τ-transitions during the traversal rather than following the completion of the main search of the model. This results in the rule allowing further simplification by itself for a model during application. After copying transitions from a target state to a source state the source state just analysed will (usually) now have new transitions to compare against removal conditions. For this reason, we add the same source state just analysed to the queue again.

Counterexample Expansion

This counterexample extraction involves recognising any τ-transitions (and consequently any removed states) which may have been removed between any step of the given counterexample. Thus, the reasoning used for designing an algorithm to compute this expansion is the same as that for expanding a counterexample given after applying observation equivalence (please refer to section 3.3.1).

Let A be the automaton to be abstracted, α be the α-marking in A's alphabet,τ be the silent event in A's alphabet, Q be a queue containing states to visit and T be a list of target states of transitions to be removed.

```
1.  if (A's alphabet does not contain α || A's alphabet does not contain τ) {
2.      return;
3.  }
4.  for each (state s in A's states) {
5.      add s to Q;
6.  }
7.  while (Q is not empty) {
8.      get state s from Q;
9.      if (s has τ-successors) {
10.         initialise T to be empty;
11.         for each (τ-successor t of s) {
12.             if (t is not α-marked) {
13.                 add t to T;
14.             }
15.         }
16.         if (T is not empty) {
17.             for each (state t  in T)  {
18.                 copy transitions originating from t to s;
19.                 remove τ transition with source s and target t;
20.             }
21.             add s to Q;
22.         }
23.     }
24. }
```

Figure 3.15: Removal of τ-Transitions Leading to Non-α States Algorithm.

3.3.8 Removal of τ-Transitions Originating From Non-α States

Removing τ-transitions which originate from non-α states is more restrictive than the previous rule discussed. Because of this, the two rules are less similar than their names suggest. The previous rule will remove a τ-transition and its target state *if* it becomes non-coreachable. In contrast, removal of τ-transitions originating from non-α states guarantees that if a τ-transition can be removed, a source state is definitely removed as well. Thus, always reducing the state space abstraction is possible. Both rules may increase the complexity of the automaton, because of the transitions which must be copied to ensure the simplified automaton is (α, ω)-nonblocking equivalent to the original version. However, removal of τ-transitions originating from non-α states will usually do this less often since this will only occur when a removed state has multiple outgoing τ-transitions.

Rule A state x that is not marked α or ω can be removed, if $x \xrightarrow{\tau} y$, and x has only τ-transitions outgoing. Incoming transitions to x must be redirected to all the τ-successor states of x (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009).

If the source and target state of a silent transition are both α-marked, the removal of α-markings rule can be applied and as a result the α-marking will be removed from the source state. Once a source state of a silent transition is not α-marked this rule has the potential to remove the source state altogether. This is evidence for why removing α-markings will enable improved simplification by other rules.

Example Figure 3.16 shows an example of the application of the removal of τ-transitions originating from non-α states rule. The transition from *S0* to *S1* labelled e_1 and e_2 shows a common way of depicting multiple transitions between the same states. The only two states of A_1 with potential for removal are *S1* and *S2*, because all other states have an α-marking or an ω-marking. *S2* does not have any outgoing τ-transitions and so does not qualify for removal. *S1* satisfies all requirements, no α-marking and no ω-marking, at least one outgoing τ-transition and all outgoing transitions are τ-transitions. Therefore, *S1* can be removed from A_1. In addition all the incoming transitions (e_1 and e_2) to *S1* must be added to the already existing incoming transitions of *S1*'s immediate τ-successors (*S2* and *S3*). A_2 shows the removal of *S1* and the subsequent redirection of transitions.

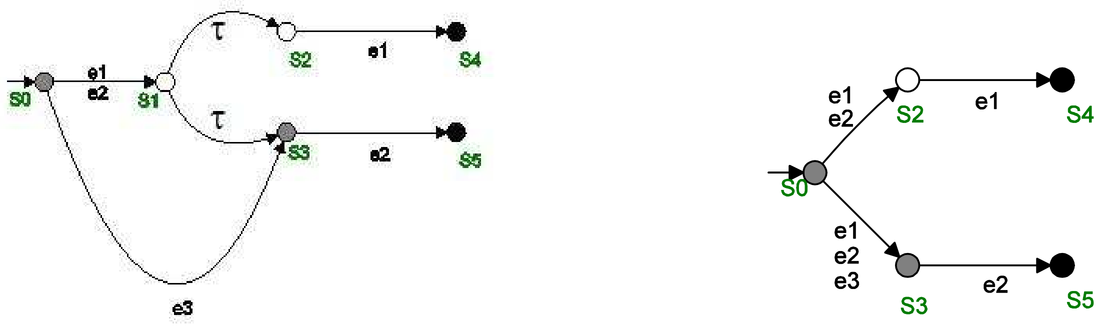

Figure 3.16: Automaton A_2 (right) shows automaton A_1 (left) after applying the Removal of τ-Transitions Originating from Non-α States rule.

Algorithm

Figure 3.17 gives an algorithm for our implementation of removal of τ-transitions originating from non-α states. This algorithm will apply the rule to every state of the automaton. The first step of the algorithm is to check that the automaton's event alphabet contains an α-marking and ω-marking. Without explicit inclusion of either marking no modification is possible. This is true since all states will be implicitly marked by one or both propositions, and the rule stipulates that a state can only be removed if the target state of a τ-transition is not α-marked and not ω-marked. Secondly, to improve efficiency the automaton's alphabet must contain a τ event to proceed; this avoids attempting to abstract when there are no τ-transitions to remove.

Let *A* be the automaton to be abstracted, α be the α-marking in *A*'s event alphabet, τ be the silent event in *A*'s alphabet, *Q* be a queue containing states to visit and *T* be a list of τ-successor states with transitions to be removed..

```
if (A's alphabet does not contain α || A's alphabet does not contain ω
        || A's alphabet does not contain τ) {
        return;
}
for each (state s in A's states) {
        if ((s is not α-marked && s is not ω-marked)
                && (s only has τ-successors)) {
                initialise T to be empty;
                for each (τ-successor t) {
                        if (t != s) {
                                add t to T;
                        }
                }
                if (T is empty) {
                        remove τ-transition with source and target s;
                } else {
                        for each (τ-successor state t  in T)  {
                                add incoming transitions to s as incoming to t;
                                remove τ-transition with source s and target t;
                        }
                        remove s from A;
                }
        }
}
```

Figure 3.17: Removal of τ-Transitions Originating From Non-α States Algorithm.

This algorithm does not employ a deep search; we are only concerned with the immediate successors of each state. We check for every state *s* that it is not α-marked and also not ω-marked. When these two requirements are satisfied the next test is to see if there are only τ-transitions outgoing from *s*. If there are any non-τ transitions outgoing, we cannot remove *s*, and must begin checking the next state. Otherwise, we proceed to adding all immediate τ-successors of *s* to a list, unless the τ-transition is a self loop; we do not want to list *s* as its own τ-successor. If this list is empty the automaton must have a τ-self loop on *s*, and we remove this transition. If the list is not empty, for every τ-successor state in the list we remove the τ-transition with *s* as source and the τ-successor state *t* as target. At the same time we redirect all incoming transitions to *s* to be incoming to *t*. Finally we remove *s* from the automaton and then begin with checking the next state.

Counterexample Expansion

The algorithm for this counterexample expansion can be done following the same steps as for expanding a counterexample given after applying the previous τ-transition removal rule (please refer to section 3.3.7). This is possible since both rules remove the same kind of information from a counterexample, simply under different constraints.

4 Experimental Results

This chapter discusses experiments which we ran to learn how to get the best out of compositional verification. The internal and final state and transition limits, heuristics and abstraction rules all contribute to success in verifying a large model. Smaller models will usually solve "easily" without the need for particular configuration of these components. Therefore, we experimented with these components in an attempt at having a systematic approach of which configurations to try first, for verifying an industrial-sized model. Prior to experimenting, we tested our compositional generalised nonblocking algorithm on a wide range of models, many of which represented real-world systems and many which were hand designed by the author to test special cases. The models used included complex industrial models and case studies taken from various application areas (Flordal & Malik, 2009).

4.1 Models

We selected a small group of the larger models used during testing which we could verify easily (i.e. without the need for very specific heuristics, state/transition limits and abstraction rules) to use for running experiments to analyse the performance of heuristics and abstraction rules. None of these models can be verified by standard approaches alone. A description of the models used for experimentation follows.

- A model of the central locking system of a BMW car. For this model a particular part of the model can be removed to get a blocking system (Flordal & Malik, 2009).
    - `verriegel4` – a four-door (α, ω)-*nonblocking* model
    - `verriegel4b` – a four-door (α, ω)-*blocking* model
- `big_bmw`, a model describing the window lift controller of a particular BMW (Malik P. , 2003).
- `fzelle`, `ftechnik` are models of case studies of two different production cells.
- Model names which start with `profisafe`, model the PROFIsafe field bus protocol.
- `rhone_alps` and all model names which contain `aip`, model an automated AIP manufacturing system.
- Model names which begin with `tbed`, model a train testbed.

In addition to these experiments, we attempted to verify a group of models which were too large to verify easily with our compositional verification implementation and have never been verified before. We managed to verify at least one of these models, SIC5 Version of AIP automated manufacturing system (Song, 2006).

4.2 Experiments

Before gathering results to be presented, preliminary experiments were conducted to find reasonable limits which would allow verification to complete for all the models we selected for future experimentation. Abstraction can be a more expensive process, in terms of time, than using a standard approach alone. Therefore, the fastest result is usually reached when a model is abstracted just enough to be of a size which can be verified using the standard approach. Thus, the intention was to find internal limits as low as possible, to avoid abstracting larger candidates than necessary. However, an internal limit can be too low if too many large candidates are not allowed to be abstracted and become blacklisted, and so the standard verification used in the final step will still not be able to verify the model. The limits used for all experiments are shown below:

- Internal state limit: 5,000
- Internal transition limit: 1,000,000

We added facilities for our implementation to compute a range of statistics during execution, including a number of statistics for every individual rule applied. The statistics for each rule accumulate each time the rule's algorithm is used. Next, we implemented an automation program, for running experiments with the models and limits we had chosen. This automation program collates all statistics ready for analysis.

4.2.1 Heuristics

Three common heuristics have previously been developed for both step one and step two of the candidate selection process (please refer to section 3.1); these heuristics are presented in (Flordal & Malik, 2006). Step one groups automata together following some heuristic, this is done to avoid considering all possible combinations of automata as candidates. Step two attempts to select the best candidate from those put forward after step one. The following definitions are for the well-known heuristics for the first step:

minT	Candidates are all automata pairs containing the automaton with the fewest transitions.
maxS	Candidates are all automata pairs containing the automaton with the most states.

mustL For each event there is a candidate which is the set of automata using that particular event.

minT and maxS are *pairing heuristics*. These heuristics choose a single automaton which satisfies some condition, e.g. minT chooses the automaton with the fewest transitions, and then pairs that automaton with every other automaton available for selection. For our implementations, if more than one automaton matches the criteria for the heuristic, then the first automaton (ordered alphabetically by name) found with the correct property is used for pairing. For example, for a model with 5 automata, if automaton *A* and automaton *B* both have the least number of transitions (10 each), automaton *A* would be paired with all others. Thus, our implementation for pairing heuristics will always produce candidates consisting of two automata. The third heuristic here, mustL, can produce candidates containing any number of automata; this is true since all automata which use some event are considered a candidate.

The three common heuristics for step two:

maxL Choose the candidate with the highest proportion of local events.
maxC Choose the candidate with the highest proportion of shared (or common) events.
minS Choose the candidate for which the product of the number of states of the automaton is smallest.

The nature of the step two heuristics mean that more than one heuristic may satisfy the condition. For example, if automaton *A* from earlier was paired with every other automaton in the model, using minT, we would have 4 candidates to choose from in step two. The highest proportion of local events might be 0.6 and multiple candidates may have this proportion. In this case, we apply the next step two heuristic, in the order shown above, to the candidates which had 0.6 as their proportion. Occasionally, all three heuristics may be applied in turn without only one candidate remaining. For this reason, we implemented a default heuristic to use. Candidate names come from their automata names appended together; our default heuristic will choose the first candidate name in a lexicographical ordering.

No known work exists which discusses the exact implementation of the above heuristics or that provides a more precise explanation of them. We implemented each heuristic in a straightforward manner, which would follow the guidelines available. After a short time experimenting with how these simple heuristics behave, we were able to refine our straightforward implementations in an attempt to make better candidate choices. This section describes the various ways we implemented these heuristics, and variations of them.

Step One Heuristics

minT One automaton is chosen which has the fewest number of transitions, and paired with each other automaton in the model, if the pair (as a candidate) satisfies the following:

- was not previously blacklisted. A candidate can be *blacklisted* if its state space exceeds internal state or transition limits at any time (please refer to section 3).
- the two automata *share* at least one event

A candidate, whose automata do not share any events, can potentially have a huge synchronous product state space since no events can be synchronised. Therefore, it is preferable to suppress any candidate which has no shared events between its automata.

Originally, we decided suppressing candidates with no local events was also preferable. A candidate, with no local events cannot have any events hidden, thus there is not great potential for abstraction. However, we found that often all events would be used by at least three automata. The consequence of this scenario is that pairing heuristics (which select candidates containing two automata) will always produce candidates without local events, and so all candidates would be suppressed. Hence, our final implementations do not suppress candidates with no local events.

maxS One automaton is chosen which has the most states, and paired with each other automaton in the model. The pair (as a candidate) must satisfy the same two conditions as for minT.

Introducing conditions on what candidates are allowable means sometimes all candidates are suppressed. In this case the automaton chosen for pairing is not considered, and we attempt to pair the next automaton which satisfies the heuristics property. This process is repeated until we find a valid candidate, or until all automata have had pairing attempted and their candidates suppressed. Usually, this will only happen once a model has been simplified greatly. Therefore, when we do not eventually find a candidate, the compositional approach is ended and we continue using the standard method for verification.

mustL For every event in the model's alphabet, a candidate is produced which consists of the automata using that event, *if* the following conditions are satisfied by the candidate:

- candidate has more than one automaton
- candidate does not include all automata of the model
- this group of automata was not already considered as a candidate
- was not previously blacklisted

There is no need to check if candidates produced using mustL share an event, this condition is satisfied by the definition of the heuristic.

Step Two Heuristics

Step two heuristics are more ambiguous than the step one heuristics implemented. Thus, we explain the various ways we calculated the proportion/value associated with each.

> **maxL** highest proportion of local events = # of local events / # of total events
> The number of local events is a count of the local events each automaton of the candidate has, *excluding* the candidate's τ-event.
> The number of total events is a count of all unique events used by each automaton of the candidate, *excluding* the candidate's τ-event.
>
> **maxLt** highest proportion of local events = # of local events / # of total events
> The number of local events is a count of the local events each automaton of the candidate has, *including* the candidate's τ-event.
> The number of total events is a count of all unique events used by each automaton of the candidate, *including* the candidate's τ-event.
>
> **maxC** highest proportion of *common* events = # of common events / # of total events
> The number of *common* events is a count of the events which all automata of the candidate use, *excluding* the candidate's τ-event.
> The number of total events is a count of all unique events used by each automaton of the candidate, *excluding* the candidate's τ-event.
>
> **maxCt** highest proportion of *common* events = # of common events / # of total events
> The number of *common* events is a count of the events which all automata of the candidate use, *including* the candidate's τ-event.
> The number of total events is a count of all unique events used by each automaton of the candidate, *including* the candidate's τ-event.

For maxL and maxC we also implemented and experimented with versions which counted the number of transitions in the automata which used the associated type of event (i.e. for maxL we counted the number of transitions labelled with a τ-event), rather than just counting the number of events. However, this computation is more complex and expensive in terms of time than the variations explained earlier, and the results were not successful enough to continue using this variation further.

The third heuristic for step two depends on the size of the state space of the synchronous product. We cannot know this value without constructing the synchronous product;

therefore we must predict a reasonable size for the state space. We compute a prediction for the size with respect to the number of local events the candidate has, based on results presented in (Shi, 2009).

minS We first compute the state space *s* by multiplying the number of states for each automaton of the candidate. We then make our prediction based on this value with respect to the number of local events.
smallest state space of synchronous product events = *s* * # of non-local events / # of total events
The number of total events is a count of all unique events used by each automaton of the candidate, *including* the candidate's τ-event. Similarly, the number of local events is a count of the events which only the automaton of the candidate use, *including* the candidate's τ-event.

The number of *non-local events* = # of total events - # of candidate's local events.

minSc We first compute the state space *s* by multiplying the number of states for each automaton of the candidate. We then make our prediction based on this value with respect to the number of shared events.
smallest state space of synchronous product events = *s* * # of common events / # of total events
The number of total events is a count of all unique events used by each automaton of the candidate, *including* the candidate's τ-event.
The number of *common events* is a count of the events which all automata of the candidate use.

4.2.2 Abstraction

Our experiments also included varying the order of the abstraction rules used and the number of times they are used. By doing this we were able to analyse which rules are more powerful and when it is best to use them or if a rule is even worthwhile. A description of the algorithms we used for implementing each of these rules is given in section 3.3.

4.3 Results

Our compositional verification algorithm has been used to check whether the models chosen for experimentation satisfy nonblocking.

Table 4.1 shows a verification using our algorithm, with minT/maxL as heuristics, of each model we used for experimentation. This is given to provide insight to what our algorithm encounters as it performs experiments. The table shows the number of automata in the model, an approximation of the size of the reachable state-space, the largest number of states and transitions encountered in a single automaton (candidate) during verification, the total number of states and transitions encountered, the total time for verification, and the result (NB denotes (α, ω)-nonblocking). Models which do not have their size specified are so large that we cannot compute its synchronous product to approximate the state space size. We can see that a very large model can be verified quickly, and the largest automaton dealt with during composition has on average less than a few thousand states.

The last row of the table shows the verification of the very large model presented in (Song, 2006), which up until now had not been verified. The property we verified for this model is one that requires multiple verifications of generalised nonblocking, and is presented as SIC Property 6 in (Leduc, Brandin, Lawford, & Wonham, 2005). Through experimentation we were able to verify this property with an internal state limit of 5000, an internal transition limit of 650000 and the heuristics used were minT and maxL. SIC5_Version_of_AIP does satisfy SIC Property 6.

Model	**Aut.**	**Size**	**Max**		**Total**		**Time (s)**	**NB**
			States	**Trans.**	**States**	**Trans.**		
big_bmw	31	3.10E+07	75	526	316	2029	0.20	y
ftechnik	36	1.20E+08	3247	25185	11633	78303	8.42	n
verriegel4	65	4.50E+10	446	6136	2669	20506	0.69	y
verriegel4b	64	6.30E+10	752	10278	3442	29110	1.19	n
rhone_alps	35		252	1142	669	2607	0.44	n
tbed_ctct	84		4876	55264	25629	227493	47.59	n
fzelle	67	3.00E+10	1672	10034	4778	23725	1.16	y
tbed_uncont	84		2620	18016	14793	80848	9.72	y
tbed_noderail	84		1820	14616	12122	75358	8.28	y
tbed_noderail_block	84		1820	14616	12308	76613	9.26	n
tbed_valid	84		4672	18016	19065	95426	13.06	y
profisafe_i4_host	28		507	17531	1359	48005	1.83	y
profisafe_i5_host	28		722	29706	1748	71973	2.94	y
SIC5_Version_of_AIP	51		8032	89582	20423	164410	22.04	

Table 4.1: Experimental results for compositional nonblocking verification.

The following two sections present more in depth experimental results.

4.3.1 Heuristic Results

We ran an experiment for all of our experimental models with each possible combination of the heuristics we implemented. The goal of these experiments was to determine the most effective heuristics for selecting candidates to compose. Better performance is indicated by a shorter run time. The heuristics used did not seem to have an impact on which order to apply the abstraction rules. Thus, the results for heuristics were gathered using a fixed ordering of the rules and only runtimes are presented.

The results of these experiments are presented in Table 4.2. The table shows the total amount of time to complete (in seconds) verification. The top row specifies the step one heuristic used, and the row below it identifies the heuristic used in step two of candidate selection. A blank cell indicates that this particular heuristic pairing, used in combination with the internal limits specified earlier, is not able to verify the model (usually due to the internal state limit being exceeded).

The step one heuristic minT was most powerful when paired with maxL, closely followed by its pairing with maxC. maxL performs significantly better than maxLt when paired with minT, and when paired with the other step one heuristic the two variations of minT perform similarly to each other. Therefore, excluding τ from the maxL computation seems to be preferable, and the same stands for maxC. This is justified by the fact that while τ may be in the event alphabet of the automata which make up a candidate, abstraction previously occurred based upon this τ-event and no further simplification is possible with these τ–transitions.

When using maxS as a step one heuristic maxC and maxCt performed similarly. maxC appears to be the best heuristic to pair with maxS, with one significant exception of verifying rhone_alps. minS, maxL and maxLt were able to verify rhone_alps over 20 times faster than maxC. minS achieved nearly as much success as maxC, however it was not able to verify two of the models and verifying verriegel4 took 221 times as long as the next slowest heuristic pairing. This problem with verifying verriegel4 also occurred when pairing minS with minT. minS's variation minSc was unable to successfully select good candidates more often than any other step two heuristic.

It can be seen that the step one heuristic mustL does not have any outright best performing combination of heuristics. One of the variations of maxL or a variation of minS seems to be preferable. minS and minSc seem to contrast one another. For example, often if minS is effective for a model, minSc will usually not be as effective and vice versa. mustL verifies much more slowly than minT. However, given that mustL was the only step one heuristic able to verify all models when paired with any of the step two heuristics we implemented, mustL has the greatest potential for being able to verify a model.

mustL can have multiple automata as a candidate, whereas minT and maxS are pairing heuristics and will only ever contain two automata as a candidate. Multiple, or even many, automata being composed at once rationalizes why verification will be slower using mustL than minT. This may also justify mustL's ability to verify the widest range of models. Having many automata composed at once may lead to a more effective ratio of local events to common events. The candidate's synchronous product may be relatively small since there are a reasonable number of events to synchronise on, and at the same time a great amount of simplification could occur since there is also a reasonable amount of local events.

maxS produces slower run times than mustL, this seems reasonable since candidates are chosen as those which may potentially have huge synchronous product state space's to construct. This also explains why maxS is the step one heuristic which most often leads to not being capable of choosing candidates that will allow verification.

The vast range in times taken for verifying verriegel4 and rhone_alps suggest that models have patterns and/or features which make a specific type of heuristic perform very well for verifying it, or perform extremely badly. This shows that future work on analysing models and categorising them in some way could be beneficial for designing and choosing heuristics which lead to verification of particular models.

Figure 4.1 shows the runtime (in seconds) for verifying each model in Table 4.2 with all possible heuristic combinations. Overall, minT paired with maxL was the most successful pairing in terms of speed. This success is shown in Figure 4.1, with minT's bars being clustered lower than the other step one heuristics. This also proves our implementation of compositional verification to be highly successful, since none of these models can be verified using standard approaches, and our algorithm verified most in less than 10 seconds when effective heuristics are used.

mustL has the greatest potential for verifying a wide range of models, given that it was the only step one heuristic able to verify all models when paired with any of the step two heuristics we implemented. Figure 4.1 shows that many of maxS's verification times were too high to be displayed, and Table 4.2 clearly showed it failed to verify a model most often. Therefore, maxS is the least preferable step one heuristic when attempting to verify a model. minSc was the least successful step two heuristic, with high run times and failing to verify a model more often than any other heuristic.

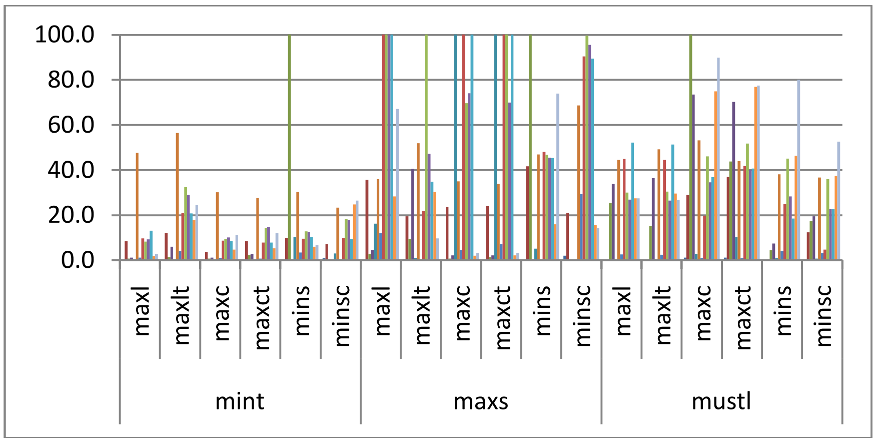

Figure 4.1: Time (s) taken for compositional verification using different heuristic combinations.

	mint						maxs						mustl					
Model	**maxl**	**maxlt**	**maxc**	**maxct**	**mins**	**minsc**	**maxl**	**maxlt**	**maxc**	**maxct**	**mins**	**minsc**	**maxl**	**maxlt**	**maxc**	**maxct**	**mins**	**minsc**
big_bmw	0.2	0.3	0.2	0.2	0.2	1.0	0.3	0.6	0.3	0.2	0.2	2.1	0.2	0.3	1.1	1.2	0.4	0.4
ftechnik	8.4	12.1	3.7	8.4	9.8	7.2	35.7	19.5	23.7	24.1	41.6	21.1	0.3	0.2	29.0	37.0	0.5	12.4
verriegel4	0.7	1.3	0.7	2.4	1032.1		2.7	9.5	0.8	1.4	2214.4		25.5	15.3	193.5	43.8	4.5	17.5
verriegel4b	1.2	5.9	1.1	2.8			4.6	40.5	2.2	2.2			33.9	36.4	73.5	70.3	7.4	19.5
rhone_alps	0.4	0.5	0.4	0.4	10.3	3.0	16.2	1.1	327.4	266.0	5.1		0.2	0.2	2.8	10.3	0.9	0.8
tbed_ctct	47.6	56.5	30.2	27.5	30.3	23.3	35.9	51.9	35.0	33.8	47.0	68.6	44.5	49.2	53.2	43.9	38.1	36.7
fzelle	1.2	4.2	1.0	0.7	3.4		12.0		4.5	7.2		29.3	2.5	2.5	1.1	0.9	4.2	3.2
tbed_uncont	9.7	20.9	8.6	7.9	9.6	9.9	113.4	21.9	138.2	136.6	48.1	90.5	44.9	44.5	19.8	41.7	24.9	4.7
tbed_noderail	8.3	32.4	9.4	14.4	12.9	18.2	149.1	136.7	69.7	144.4	46.9	99.6	30.1	30.4	46.1	51.8	45.1	36.0
tbed_noderail_block	9.3	29.1	10.1	14.8	12.6	17.9	145.3	47.1	74.0	69.9	45.5	95.6	26.9	26.4	34.5	40.0	28.3	22.6
tbed_valid	13.1	20.8	8.6	7.8	10.3	9.4	127.0	34.8	133.8	133.0	45.4	89.5	52.2	51.3	36.8	40.7	18.4	22.6
profisafe_i4_host	1.8	17.8	4.8	5.3	6.0	24.8	28.3	30.3	2.0	2.2	16.0	15.6	27.5	29.7	74.9	76.9	46.4	37.4
profisafe_i5_host	2.9	24.5	11.3	12.0	6.8	26.4	67.1	9.7	3.3	3.4	73.9	14.3	27.4	26.8	89.8	77.4	80.0	52.6

Table 4.2: Experimental results of time taken (in seconds) for different heuristic combinations to complete compositional verification.

4.3.2 Abstraction Results

Abstraction is of upmost importance for compositional verification, given that this is where all simplification occurs, in turn allowing us to verify very large models. We ran many experiments altering the order we apply the abstraction rules. Given that the heuristics used for selecting candidates to compose do not seem to have a noticeable effect on the abstraction rule ordering, all of the results presented in this section were from experiments using the heuristic pairing we found to be most effective from evaluating the results of the last section's experiments (step one using minT, step two using maxL).

The results presented are for a couple of different orderings which were most successful for applying the abstraction rules. As well as altering the order of rules, we experimented with applying some rules multiple times during one abstraction process with other rules applied in between. However, the results showed applying a rule more than once for a candidate was not worthwhile. The simplification from the second application of a rule was so minimal that the time taken for the extra application outweighed the simplification gain. Thus, further experimentation was not done with this and no figures are presented.

Results were gathered for all models chosen for experimentation. Our algorithm iteratively applies the rules, so each rule will nearly always be applied more than once. Thus the statistics recorded for rule performance is an accumulation of each time the rule is applied to a model. These accumulate values were collected for every model we chose to experiment with, we then calculated averages of how the rule performed for the purpose of presenting. Table 4.4 presents experimental results for a few different orderings of the abstraction rules. The table lists the rules in the order they were applied by the algorithm and includes data for the average time the rule uses during verification of a model, the average probability that the rule will make a simplification, the average percentage the state space is reduced by and the average percentage the number of transitions is reduced by.

While observation equivalence is the most expensive rule in terms of time, it is also by far the most powerful rule in terms of simplification. Observation equivalence has a significantly higher probability of a reduction occurring than any other rule, and the reduction is greater than the other rules achieve. Table 4.3 shows that when only applying observation equivalence during abstraction we were able to verify all of the models chosen for experimentation. However, it should be noted that we want to be able to verify models even larger than these. Therefore, the additional simplification other rules can provide is necessary for much larger models.

Model	Result	Time (s)
big_bmw	y	0.20
ftechnik	n	9.00
verriegel4	y	0.74
verriegel4b	n	1.19
rhone_alps	n	0.51
tbed_ctct	n	49.83
fzelle	y	1.32
tbed_uncont	y	9.86
tbed_noderail	y	8.53
tbed_noderail_block	n	9.75
tbed_valid	y	13.73
profisafe_i4_host	y	2.17
profisafe_i5_host	y	3.19

Table 4.3: Experimental results for only applying observation equivalence.

Removal of α–markings and removal of ω–markings do not make reductions in the number of transitions or in the state space, rather they pave the way to making other rules more powerful. Therefore, it is preferable to apply these two rules early on; this is how the rules complement one another. While some rules' results do not appear significant enough to justify applying them, we have found that when we do not apply them other rules are affected. Another rule which does not make reductions in our results is removal of non-coreachable states, yet it does consume execution time. The reason for this is these models are standard nonblocking problems being treated as generalised nonblocking. Thus, all states have an α–marking to begin with, and removal of these markings can only occur when a state is still (α, ω)-coreachable. Therefore, this rule cannot achieve any simplification when we are treating a standard nonblocking problem as generalised.

The increase in the number of transitions after applying removal of τ–transitions leading to non-α states was justified in section 3.3.7). Removing a τ–τ–transition using this rule requires redirecting all the outgoing transitions from the τ–τ–transition's target state to its source state; this results in an increase in the number of transitions, when there are multiple outgoing transitions from the target. The impact of this is that future processing will take longer with a greater number of transitions to search. We experimented with not applying removal of τ–transitions leading to non-α states; this lead to nearly all of our experimental models being verified more quickly. However, we are interested in being able to verify models even larger than our experimental ones. Thus, we would like to know whether the simplification provided by this rule is needed to verify vary large models. We were able to verify the previously unsolved model SIC5_Version_of_AIP more than 30% faster when we did not apply removal of τ–transitions leading to non-α states. This result was promising. As expected further experimentation showed that not applying this rule will result in more efficient verification, however it does not increase the capability of verifying even larger models. Thus, future work could be directed towards uncovering abstraction rules which do not have such great consequences.

Removal of τ-transitions originating from non-α states appears to make no reduction at all, however a tiny reduction is usually achieved. When the state space and number of transitions is so large, the reduction is smaller than what two decimal points can represent. Because of such a tiny reduction percentage, we experimented with not applying this rule. However, the increase in speed of verification was so insignificant that the small reductions seem worthwhile to increase the likeliness of verifying extremely large models.

The two different orderings show that the order the rules are applied do not seem to have a significant effect on how powerful they are in a positive or negative way, yet we have said we have seen proof that the rules complement each other. Presumably, our results show that the order is irrelevant, due to the fact that the abstraction rules are applied many times during verification and our rule statistics are accumulated. Thus, if a rule is applied third, and the rule applied afterwards has the potential to make the rule in position 3 more powerful, the rule at position 3 will get another chance to perform this abstraction in the next cycle, making it appear that the ordering does not matter. The order in which rules are applied would probably have an effect if all rules were applied only once, but since compositional verification is an iterative algorithm the ordering becomes less important.

Rule	Time (s)	Prob.	State Red. %	Trans. Red. %
τ–Loop Removal	0.30	0.09	0.05	0.06
Observation Equivalence	2.56	0.61	0.29	0.31
Removal of α–markings	0.08	0.35	N/A	N/A
Removal of ω–markings	0.08	0.01	N/A	N/A
Removal of Non-coreachable States	0.09	0.00	0.00	0.00
Determinisation of Non-α States	1.85	0.11	0.07	0.08
Removal of τ–Transitions Leading to Non-α States	0.09	0.14	0.00	-0.24
Removal of τ–Transitions Originating From Non-α States	0.06	0.00	0.00	0.00
τ–Loop Removal	0.3	0.09	0.05	0.06
Observation Equivalence	2.56	0.61	0.29	0.31
Removal of α–markings	0.08	0.35	N/A	N/A
Removal of ω–markings	0.08	0.01	N/A	N/A
Determinisation of Non-α States	1.88	0.11	0.08	0.09
Removal of τ-Transitions Originating From Non-α States	0.06	0.05	0.00	0.00
Removal of τ-Transitions Leading to Non-α States	0.08	0.13	0.00	-0.23
Removal of Non-coreachable States	0.08	0.00	0.00	0.00

Table 4.4: Experimental results for average performance of abstraction rules.

5 Conclusions

The first implementation for compositionally verifying generalised nonblocking has been presented. Our implementation has been integrated with the Supremica (Supremica, 2010)(Akesson, Fabian, Flordal, & Malik, 2006) toolkit, thus it is available to use. Our experimental results have shown the success of this implementation with many models (which are too large to be verified using standard methods for verification) being verified in less than 10 seconds. Further evidence of its success was in verifying a model which has never been verified before.

Experimental results showed that the heuristics used for selecting candidates to compose have more impact on the speed at which a model is verified, than the order of abstraction rules used. Our heuristic results also showed that subtle difference in the way heuristics are calculated can make a significant difference to that heuristics success in being able to verify a model (e.g. maxL and maxLt). Therefore, in future we could improve on the speed for verifying models, and on the size of models which can be verified, by studying other heuristics to use. We have also seen that the ability to categorise qualities of a model, could assist in choosing which heuristics would be most effective for verifying that model.

Experimental results proved theories in (Leduc & Malik, Seven abstraction rules preserving generalised nonblocking, 2009) correct that the abstraction rules do complement each other and the order of which they are applied can positively affect the potential of another rule. However, since compositional verification will repeatedly abstract a model the order in which rules are applied does not seem to have a noticeable effect. Observation equivalence is noticeably the most powerful rule for simplification. Further work towards developing abstraction rules which can achieve reductions similar to this would be tremendously beneficial for verifying the largest of models.

References

Akesson, K., Fabian, M., Flordal, H., & Malik, R. (2006). Supremica - an integrated environment for verification, synthesis and simulation of discrete event systems. *Proc. 8th Workshop on Discrete Event Systems (WODES'06)*, (pp. 384-385). Ann Arbor, USA.

Berard, B., Bidoit, M., Finkel, A., Laroussine, F., Petit, A., Petrucci, L., et al. (2001). *Systems and Software Verification.* Springer.

Brandin, B. A. (2000). *VALID Tool Set: Validation of Software Applications*. Retrieved July 2010, from 5th Workshop on Discrete Event Systems (WODES 2000): http://www.diee.unica.it/giua/WODES/WODES2000/brandin.html

Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). Model Checking. *MIT Press* .

Eloranta, J. (1991). Minimizing the Number of Transitions with Respect to Observation Equivalence. *BIT , 31* (4), 397-419.

Fernandez, J.-C. (1990). An Implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming* , 13:219-236.

Flordal, H., & Malik, R. (2009). Compositional verification in supervisory control. *SIAM J. Control and Optimization , 48* (3), 1914-1938.

Flordal, H., & Malik, R. (2006). Modular nonblocking verification using conflict equivalence. *Proceedings of the 8th International Workshop on Discrete Event Systems, WODES'06*, (pp. 100-106). Ann Arbor, MI, USA.

Hoare, C. A. (1985). *Communicating Sequential Processes.* Prentice-Hall.

Leduc, R., & Malik, R. (2009). A compositional approach for verifying generalised nonblocking. *Proceedings of 7th International Conference on Control and Automation, ICCA '09*, (pp. 448-453). Christchurch, NZ.

Leduc, R., & Malik, R. (2008). Generalised Nonblocking. *Proceedings of the 9th International Workshop on Discrete Event Systems, WODES'08*, (pp. 340–345). Sweden.

Leduc, R., & Malik, R. (2009). *Seven abstraction rules preserving generalised nonblocking.* Working Paper: 07/2009, The University of Waikato, Department of Computer Science, Hamilton, NZ.

Leduc, R., Brandin, B., Lawford, M., & Wonham, W. (2005). Hierarchical Interface-based Supervisory Control - Part I: Serial Case. *IEEE Transactions on Automatic Control , 50* (9), 1322–1335.

Malik, P. (2003). *From Supervisory Control to Nonblocking Controllers for Discrete Event Systems.* University of Kaiserslautern, PhD Thesis, Kaiserslautern, Germany.

Malik, R., & Ware, S. (2008). The use of language projection for compositional verification of discrete event systems. *Proc. 9th International Workshop on Discrete Event Systems (WODES'08)*, (pp. 322-327). Sweden.

Milner, R. (1980). *A Calculus of Communicating Systems* (Vol. 92 of LNCS). Springer-Verlag.

Ramadge, P. J., & Wonham, W. M. (1989, January). The Control of Discrete Event Systems. *Proc. IEEE , vol. 77, 1*, 81-98.

Shi, J. (2009). *Selection of Components in Compositional Verification of Safety Properties.* MSc Thesis, The University of Waikato, Department of Computer Science, Hamilton, NZ.

Song, R. (2006). *Symbolic Synthesis and Verification of Hierarchical Interface-Based Supervisory Control, M.A.Sc. Thesis.* McMaster University, Department of Computing and Software.

Supremica. (2010). The official website for the Supremica project. *www.supremica.org* .

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing , 1* (2), 146-160.

Ware, S. (2007). *Modular Finite-State Machine Analysis.* Report of an Investigation, The University of Waikato, Department of Computer Science, Hamilton, NZ.

Wen, Y., Wang, J., & Qi, Z. (2004). Reverse observation equivalence between labelled state transition systems. *Proceedings of 1st International Colloquium on Theoretical Aspects of Computing, ICTAC '04*, (pp. 204-219). Guiyang, China.