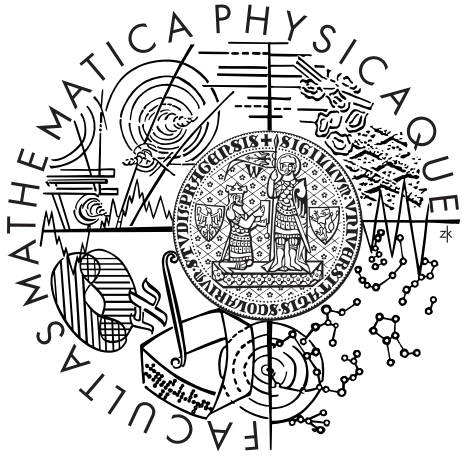
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jakub Daniel

Analysis of Interface Automata with On-Demand Replication

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Jan Kofroň, Ph.D.
Study programme: Informatics
Specialization: Software Engineering

Prague 2013

I would like to sincerelly thank my supervisor RNDr. Jan Kofroň, Ph.D. for his time, helpful advices, guidance and constructive criticism.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date April 8, 2013 signature of the author

Název práce: Analysis of Interface Automata with On-Demand Replication

Autor: Jakub Daniel

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: RNDr. Jan Kofroň, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt:

Interface automat je model chování softwarové komponenty založený na konečných automatech. Popisuje její poskytovaná rozhraní neboli podporované použití a požadovaná rozhraní neboli použití ostatních komponent. Značný počet komponent může být použit paralelně bez omezení úrovně paralelismu. Není nutné, aby se model pokoušel zachytit tuto neomezenost. Alternativním přístupem je umožnit zvyšování úrovně paralelismu na vyžádání. Tato práce na teoretické úrovni analyzuje a navrhuje konečnou podobu operace k zajištění tohoto typu replikace s cílem umožnit konstrukci modelů libovolné úrovně paralelismu v určitých částech jejich chování.

Klíčová slova: Interface automat, paralelismus, replikace, modely chování

Title: Analysis of Interface Automata with On-Demand Replication

Author: Jakub Daniel

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Jan Kofroň, Ph.D., Department of Distributed and Dependable Systems

Abstract:

Interface automaton is a model of software component behaviour based on finite state machines. It describes component's provided interface, the supported usage, and required interface, the usage of other components. A considerable number of components can be used in parallel with no bound on the level of parallelism. It is not necessary for the model to attempt to capture such unboundedness. An alternative approach is to allow an increment of the level of parallelism on-demand. This thesis analyses on a theoretical level and proposes a final form of an operation to perform such replication to allow creation of models of an arbitrary level of parallelism of certain parts of its behaviour.

Keywords: Interface automaton, parallelism, replication, behaviour models

Contents

Introduction

In software analysis and verification there exist different ways of modelling behaviour of software components. Interface automata are one such modelling technique that captures temporal interface requirements and provisions of components. By doing so they allow detection of inconsitencies in models and by extension incompatibilities of the modelled components themselves.

Among all the components there are many that may be used in parallel. Models of such systems are usually constructed for a fixed number of threads with only a limited possibility of automation of the construction. This thesis attempts to advocate on a theoretical level an approach dealing with a more general solution based on an extension of an accepted formalism of the interface automata as it will be described in one of the following chapters. The key distinction from other approaches is that chainable operations will be provided to allow construction of automata of arbitrary degrees of parallelism while harnessing possibilities of optimisations of the product on the run provided by their iterative nature. And doing so while still being able to operate the results as standard interface automata and subject them to further parallelisation.

Attention will be paid to client-server architectures with multiple isolated clients communicating with single server in parallel. The actual scale of positive impact is, however, expected to be larger as there will be only a minimum of assumptions made about the concrete type of architecture. Therefore other types of components may benefit from the proposed operations and constructions. This particular setting serves as the most natural motivation example only.

Parallelising an interface automaton amounts to guaranteeing that all possible states and thread schedulings are covered by the resulting automaton. Different views of what needs to be captured exist depending on communication specifics involved in the composite system and type of abstraction applied for the creation of the model. First the least restricted approach will be pursued and after introduction of the basic principle a connection to operations that are in a way related to modeling multi-threading in interface automata will be made. The vision of optimisation possibilities may then provide further justification of the proposed approach and its superiority to a naive alternative suffering from over-approximation drawbacks.

The thesis is stuctured in the following manner. First a difference between modelling single-threaded compositions of components and a natural extension to multi-threaded compositions is explained on an abstracted example in Chapter 1. After demonstrating the need for operations producing automata with replicated

parts of interfaces a formal definition of the apparatus is given, starting with basic definitions established in already existing materials (Chapter 2) and ending with definitions fundamental to this thesis (Chapters 2 and 3). In the latter Chapter a basic operation introducing new thread to an automaton is provided as well. Next its properties are compared to an automaton constructed with standard operations with aim to model parallel behaviour. In the process questionable properties of the resulting models are identified and are addressed one by one in Chapters 4 and 5. As a result a new operation is described and its effects are illustrated in examples in Chapter 5. While these examples present parts of models interesting from the theoretical point of view, the practical application and target technologies used for implementation of modelled systems are provided in Chapter 6. Chapter 7 relates the chosen approach to existing alternatives and their formal background. In the end Chapter 8 suggests directions of further analysis and extensions.

1. Setting

Both the development and verification of modern software systems often make use of models. In fact the two activities are closely related. Models in general can be used to express abstracted designs of software units either with the aim to generate the concrete detailed implementation afterwards or to allow reasoning about correctness or other properties of a supplied implementation in a much more fitting domain avoiding the need to consider irrelevant characteristics of the modelled situation.

Interface automata are a well-established accepted formalism for specifying behaviour models of systems. The increase of complexity of contemporary systems supports the convenience of having such models for these purposes. Complex architectures often employ components that are composed together to produce subsystems and by extension the complete system. Individual components can be described by their required and provided interfaces along with assumptions about their use. An interoperation of two or more components is captured by composition of their models during which required interfaces of one are bound to matching provided interfaces of the other. In its generality the formalism does not provide a method of binding a concrete model with models of multiple instances of interchangable components as it is required for modelling parallel access of the mentioned components to the shared component. Although such models could be constructed manually, it would not be a scalable solution to do so. Therefore there is a need for a solution that could easily be automated.

Hereafter we would like to focus on manipulation with models rather than their precise correspondence to the modelled implementation. Despite the interface automata by themselves do not deal with parallelism in particular, they can be used to capture parallel execution of a fixed degree. Components programmed in common programming languages, however, often do not impose a concrete limit to the number of threads. In fact there usually exists a single definition of a code snippet or method that is being executed in numerous threads the number of which is not known in advance.

As it has been already mentioned, the supporting example for the effort dedicated to analysis of parallelism within interface automata is a client-server architecture, a widely spread pattern utilised by for example web services. A single-threaded server could often be described using a loop in which it processes requests and returns responses. If there were one client or multiple clients the requests of which were serialised discretely, so that all responses had been sent before the next request was made, the model of the server could feature a single

thread. This, however, describes only a limited number of servers in practice and due to the growth of importance of parallelism for not only scalability the need for more profound solution arises. In real servers processing of multiple requests received in parallel is desired and is the reason for attempting to automatically increase number of threads of the server with the least effort required from the component's designer to describe the automaton in question.

An interface requirements of a simple single-threaded server, provided and required actions and their mutual sequencing, could be modelled by a labelled state transition system as it is illustrated in Figure 1.1. Individual transition labels can be divided into input, output and internal actions denoted by ?, ! and ; respectively.

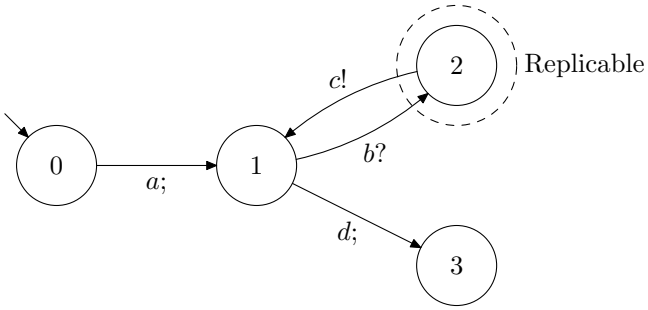


Figure 1.1: Simplified server pattern

All interactions with the modelled component start in the initial state 0 depicted by a node with an oriented sourceless edge in Figure 1.1. There is only one transition from 0 which models an action a being performed internally, be it some server initialisation. From the state 1 it is either possible to terminate the server by pursuing an edge labelled with d or accept a request b from a hypothetical client followed by response c being emitted.

To be able to model parallel executions using interface automata it is required that each transition separately is atomic. If this condition is satisfied any parallel execution is equivalent to one serialised execution which interleaves transitions from different threads.

A cartesian product of n copies (replicas) of sets of states forms a lattice whose each dimension corresponds to execution in one particular copy of the replicated section and any walk through this lattice captures one interleaving of transitions.

We would expect the model of two threaded variant of the previously mentioned server to look like Figure 1.2.

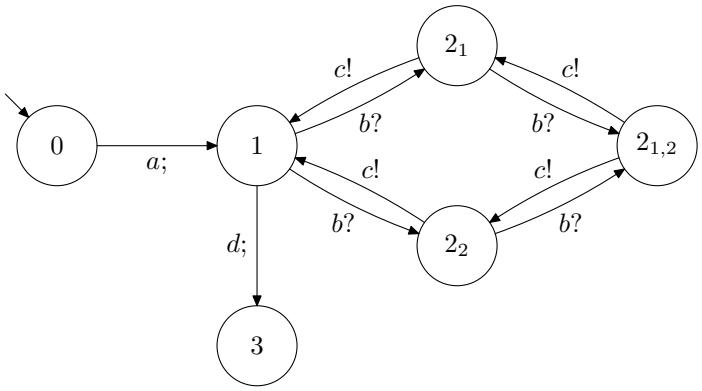


Figure 1.2: Parallelised server

A nondeterministic choice of advancement from a particular state models scheduling possibilities. In Figure 1.2 it can be seen that from state 1 there originate two transitions that correspond to exactly one thread advancing. In both the states 2_1 and 2_2 there is an action $b?$ enabled, a transition from the state labelled with the action exists, and that corresponds to an advancement of the other thread than the one before. In the state $2_{1,2}$ the two threads reached their local state corresponding to state 2 of the original single-threaded model. Alternatively we could require that each thread can move to the termination state 3 by replicating it for both the threads, the resulting automaton would be a bit too large and inappropriate for demonstration so far, although in practice it is a matter of what is expected behaviour of the component after replication.

The above examples are in fact interface automata, whose formal definition will follow in the next chapter.

2. Interface automata

Interface automata are a formalism modelling temporal aspects of input-output behaviour of components. Individual paths within an automaton form assumptions about expected event sequences in a use of a component. Models of composed systems then can be subjected to verification of compliance with assumptions made by individual components. The main focus is on open systems, i.e. systems that can be further composed with other components, which is why an optimistic approach is usually used to determine component compatibility.

Composition operation, inspired by development of component-based systems in practice, is an act which interlinks two components so that their shared parts of interfaces are matched together and synchronises executions in the two components at the point of the link, which closely corresponds to emitting and absorbing an event or in particular to calling and executing a method in modern programming languages.

While pessimistic view would consider two components incompatible in case a state where an action emitted by one component is not expected by the other component is reachable in an environment, the optimistic view reflects reality more closely by pronouncing two automata compatible if there exists an environment that ensures all executions avoid reaching such illegal states. It would be overly restrictive to disallow such a composition that can by proper handling result in an error-free system.

2.1 Basic definitions

All the definitions in this section were taken from the article by Alfaro and Henzinger [1].

Definition. *Interface automaton* $\mathcal{P}$ is $\left(V_{\mathcal{P}}, V_{\mathcal{P}}^{init}, A_{\mathcal{P}}^{I}, A_{\mathcal{P}}^{O}, A_{\mathcal{P}}^{H}, \mathcal{T}_{\mathcal{P}}\right)$ where

$V_{\mathcal{P}}$	is a set of states.
$V_{\mathcal{P}}^{init}$	is a set of initial states.
$A_{\mathcal{P}}^{I}$	is a set of input actions.
$A_{\mathcal{P}}^{O}$	is a set of output actions.
$A_{\mathcal{P}}^{H}$	is a set of hidden actions.
$\mathcal{T}_{\mathcal{P}}$	is a set of transitions from states to states performing an action.

The following must hold.

$$
\begin{aligned}
V_{\mathcal{P}}^{init} &\subseteq V_{\mathcal{P}} \\
\left|V_{\mathcal{P}}^{init}\right| &\leq 1 \\
A_{\mathcal{P}}^{I} \cap A_{\mathcal{P}}^{O} &= \emptyset \\
A_{\mathcal{P}}^{O} \cap A_{\mathcal{P}}^{H} &= \emptyset \\
A_{\mathcal{P}}^{H} \cap A_{\mathcal{P}}^{I} &= \emptyset \\
\mathcal{T}_{\mathcal{P}} &\subseteq V_{\mathcal{P}} \times A_{\mathcal{P}} \times V_{\mathcal{P}}
\end{aligned}
$$

We will further use the following designations.

$$
\begin{aligned}
A_{\mathcal{P}} &= A_{\mathcal{P}}^{I} \cup A_{\mathcal{P}}^{O} \cup A_{\mathcal{P}}^{H} \\
A_{\mathcal{P}}(u) &= \{a \mid \exists u' : (u, a, u') \in \mathcal{T}_{\mathcal{P}}\} \\
A_{\mathcal{P}}^{I}(u) &= A_{\mathcal{P}}(u) \cap A_{\mathcal{P}}^{I} \\
A_{\mathcal{P}}^{O}(u) &= A_{\mathcal{P}}(u) \cap A_{\mathcal{P}}^{O} \\
A_{\mathcal{P}}^{H}(u) &= A_{\mathcal{P}}(u) \cap A_{\mathcal{P}}^{H}
\end{aligned}
$$

The motivation behind introduction of components is their composition into larger systems. But obviously not all pairs of components can be used to produce a meaningful composite. The minimum that needs to be satisfied for two automata that are about to be composed into one is that they agree on meaning of individual actions. That is if they share an action then one of the automata needs to consider it an input and the other an output action.

Definition. Two interface automata $\mathcal{P}$ and $\mathcal{Q}$ are *composable* if and only if

$$
\left.
\begin{array}{lcl}
A_{\mathcal{P}}^{H} & \cap & A_{\mathcal{Q}} \\
A_{\mathcal{P}} & \cap & A_{\mathcal{Q}}^{H} \\
A_{\mathcal{P}}^{I} & \cap & A_{\mathcal{Q}}^{I} \\
A_{\mathcal{P}}^{O} & \cap & A_{\mathcal{Q}}^{O}
\end{array}
\right\} = \emptyset
$$

To simplify formulations in future definitions it is helpful to bring up a designation for actions that have a special meaning for composition.

Definition. Let $shared(\mathcal{P}, \mathcal{Q}) = A_{\mathcal{P}} \cap A_{\mathcal{Q}}$ for any two interface automata $\mathcal{P}$ and $\mathcal{Q}$. Notice that for two composable automata this is exactly $\left(A_{\mathcal{P}}^{I} \cap A_{\mathcal{Q}}^{O}\right) \cup \left(A_{\mathcal{P}}^{O} \cap A_{\mathcal{Q}}^{I}\right)$.

The composition of two interface automata synchronises executions of the two captured systems at point where one automaton awaits an input action and the other provides it. The rest of the transitions are carried out independently.

Definition. For two composable automata $\mathcal{P}$ and $\mathcal{Q}$ let $\mathcal{P} \otimes \mathcal{Q}$ denote their *composition* defined by:

$$
\begin{array}{rcl}
V_{\mathcal{P}\otimes\mathcal{Q}} & = & V_{\mathcal{P}} \times V_{\mathcal{Q}} \\
V^{init}_{\mathcal{P}\otimes\mathcal{Q}} & = & V^{init}_{\mathcal{P}} \times V^{init}_{\mathcal{Q}} \\
A^{I}_{\mathcal{P}\otimes\mathcal{Q}} & = & \left(A^{I}_{\mathcal{P}} \cup A^{I}_{\mathcal{Q}}\right) \setminus shared(\mathcal{P},\mathcal{Q}) \\
A^{O}_{\mathcal{P}\otimes\mathcal{Q}} & = & \left(A^{O}_{\mathcal{P}} \cup A^{O}_{\mathcal{Q}}\right) \setminus shared(\mathcal{P},\mathcal{Q}) \\
A^{H}_{\mathcal{P}\otimes\mathcal{Q}} & = & A^{H}_{\mathcal{P}} \cup A^{H}_{\mathcal{Q}} \cup shared(\mathcal{P},\mathcal{Q}) \\
\mathcal{T}_{\mathcal{P}\otimes\mathcal{Q}} & = & \{((u,v),a,(u',v))|(u,a,u') \in \mathcal{T}_{\mathcal{P}} \wedge a \notin shared(\mathcal{P},\mathcal{Q}) \wedge v \in V_{\mathcal{Q}}\} \cup \\
 & & \{((u,v),a,(u,v'))|(v,a,v') \in \mathcal{T}_{\mathcal{Q}} \wedge a \notin shared(\mathcal{P},\mathcal{Q}) \wedge u \in V_{\mathcal{P}}\} \cup \\
 & & \{((u,v),a,(u',v'))|(u,a,u') \in \mathcal{T}_{\mathcal{P}} \wedge (v,a,v') \in \mathcal{T}_{\mathcal{Q}} \wedge a \in shared(\mathcal{P},\mathcal{Q})\}
\end{array}
$$

It is apparent that one automaton may emit an action in a state where the other automaton does not accept it. Such states are called illegal.

Definition. Let $illegal(\mathcal{P},\mathcal{Q})$ denote

$$
\left\{ (u,v) \in V_{\mathcal{P}} \times V_{\mathcal{Q}} \middle| \exists a \in shared(\mathcal{P},\mathcal{Q}) \wedge \begin{pmatrix} a \in A^{I}_{\mathcal{P}}(u) \wedge a \notin A^{O}_{\mathcal{Q}}(v) \\ \vee \\ a \in A^{O}_{\mathcal{P}}(u) \wedge a \notin A^{I}_{\mathcal{Q}}(v) \end{pmatrix} \right\}
$$

Definition. We call an interface automaton $\mathcal{Q}$ an *environment* for an interface automaton $\mathcal{P}$ if

- $\mathcal{Q}$ is composable with $\mathcal{R}$,
- $\mathcal{Q}$ is not empty ($V^{init}_{\mathcal{Q}} \neq \emptyset$),
- $A^{I}_{\mathcal{Q}} = A^{O}_{\mathcal{P}}$,
- $illegal(\mathcal{P},\mathcal{Q}) = \emptyset$.

Definition. Given two composable interface automata $\mathcal{P}$, $\mathcal{Q}$ a *legal environment* $\mathcal{R}$ for $\mathcal{P} \otimes \mathcal{Q}$ is such that no $illegal(\mathcal{P},\mathcal{Q}) \times V_{\mathcal{R}}$ is reachable in $(\mathcal{P} \otimes \mathcal{Q}) \otimes \mathcal{R}$. If such environment exists we call $\mathcal{P}$ and $\mathcal{Q}$ to be *compatible*.

Although it may be considered obvious, let a cross product be defined, for completness and to avoid misinterpretations, as follows

Definition. For two interface automata $\mathcal{P}$, $\mathcal{Q}$ satisfying

$$
\left.
\begin{array}{rcl}
A^{I}_{\mathcal{P}} & \cap & A^{O}_{\mathcal{Q}} \\
A^{O}_{\mathcal{P}} & \cap & A^{I}_{\mathcal{Q}} \\
\left(A^{I}_{\mathcal{P}} \cup A^{O}_{\mathcal{P}}\right) & \cap & A^{H}_{\mathcal{Q}} \\
\left(A^{I}_{\mathcal{Q}} \cup A^{O}_{\mathcal{Q}}\right) & \cap & A^{H}_{\mathcal{P}}
\end{array}
\right\} = \emptyset
$$

we define cross product $\mathcal{P} \times \mathcal{Q}$ by

$$\begin{aligned}
V_{\mathcal{P}\times\mathcal{Q}} &= V_{\mathcal{P}} \times V_{\mathcal{Q}} \\
V^{init}_{\mathcal{P}\times\mathcal{Q}} &= V^{init}_{\mathcal{P}} \times V^{init}_{\mathcal{Q}} \\
A^{I}_{\mathcal{P}\times\mathcal{Q}} &= A^{I}_{\mathcal{P}} \cup A^{I}_{\mathcal{Q}} \\
A^{O}_{\mathcal{P}\times\mathcal{Q}} &= A^{O}_{\mathcal{P}} \cup A^{O}_{\mathcal{Q}} \\
A^{H}_{\mathcal{P}\times\mathcal{Q}} &= A^{H}_{\mathcal{P}} \cup A^{H}_{\mathcal{Q}} \\
\mathcal{T}_{\mathcal{P}\times\mathcal{Q}} &= \{((u,v),a,(u',v))|(u,a,u') \in \mathcal{T}_{\mathcal{P}} \wedge v \in V_{\mathcal{Q}}\} \cup \\
&\quad \{((u,v),a,(u,v'))|(v,a,v') \in \mathcal{T}_{\mathcal{Q}} \wedge u \in V_{\mathcal{P}}\}
\end{aligned}$$

This way the result is again an interface automaton and any $\mathcal{P}$ can be powered to any order.

2.2 Synchronisation

Because this thesis aims at examining possibilities of multi-threaded systems, it is appropriate to address concepts used in parallel computations. Often for a component there exist execution paths that cannot be interleaved without leading to race conditions. Multi-threaded executions tend to make use of synchronisation to enforce only permissible interleaving.

The notion of synchronisation is not incorporated in standard interface automata which may lead to unrealistic models or spurious claims of incompatibility since acceptance of a synchronised method invocation in one thread may not be permitted until all other threads leave the corresponding critical section.

The following proposes an extension to standard interface automata that allow a developer to annotate the component accordingly to the synchronisation requirements. The intuition tells us that some of the states of the modelled system are somehow related and may be considered in a group. We will call such groups *sections* and they may be seen as blocks in programming languages or some fractions of computation that share a common property of some kind. One of such properties is mutual exclusion. In modern languages there are usually ways to mark a specific piece of code accessible only to an owner of an exclusive privilege, in other languages there is most likely another way to achieve the same effect by using a library.

Definition. *Synchronisation annotation* $(\mathcal{S}_V, \mathcal{C}_V)$ for a set of states V is defined by:

$$\mathcal{S}_V \subseteq V \times V : \begin{cases} \forall v \in V : (v,v) \in \mathcal{S}_V, \\ \forall v_1, v_2 \in V : (v_1,v_2) \in \mathcal{S}_V \Leftrightarrow (v_2,v_1) \in \mathcal{S}_V, \\ \forall v_1, v_2, v_3 \in V : (v_1,v_2) \in \mathcal{S}_V \wedge (v_2,v_3) \in \mathcal{S}_V \Rightarrow (v_1,v_3) \in \mathcal{S}_V, \end{cases}$$

$$\mathcal{C}_V \subseteq \left\{ [v]_{\mathcal{S}_V} \middle| \forall v \in V \right\},$$

$$Exclude_V(V_1, V_2) \Leftrightarrow \exists v_1 \in V_1 \wedge \exists v_2 \in V_2 : [v_1]_{\mathcal{S}_V} \in \mathcal{C}_V \wedge [v_2]_{\mathcal{S}_V} \in \mathcal{C}_V \wedge [v_1]_{\mathcal{S}_V} = [v_2]_{\mathcal{S}_V},$$

where $\mathcal{S}_V$ denotes a relation of being part of the same section, $\mathcal{C}_V$ an arbitrary subset of critical sections and $Exclude_V$ relating sets of states that conflict with each other at least in one pair of states.

By letting $\mathcal{C}_V = \emptyset$ we can manipulate interface automata as if there was no notion of synchronisation.

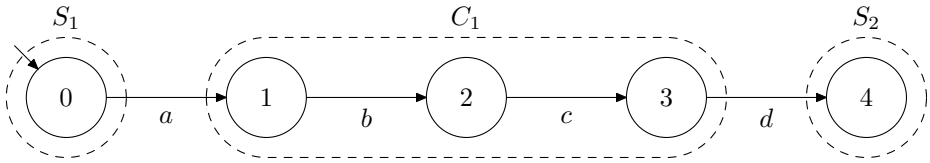


Figure 2.1: Annotated interface automaton

Example 1. The annotation depicted in Figure 2.1 corresponds to

$$\mathcal{S} = \{(0,0), (1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3), (4,4)\},$$
$$\mathcal{C} = \{\{1,2,3\}\}.$$

In this simple example it can be seen that *Exclude* relates sets containing states from the same critical section. Suppose we had a multi-threaded automaton, whose states encode states of individual threads. If we were about to add yet another thread *Exclude* could help us determine whether a compound state was valid, not allowing two threads to be found in the same exclusive state, in the resulting automaton.

As it may have been slightly indicated in the previous example, to be able to take synchronisation into account it is important that states of a multi-threaded automaton can be decomposed into original substates.

Definition. *State decomposition function* or simply *decomposition function* $D_{V_1,V}$ shall denote a mapping

$$D_{V_1,V} : V \to \mathcal{P}(V_1)$$

whose particular images depend greatly on the context in which the domain V was constructed. The intuition behind such functions should be that

$$\forall v \in V_1 : D_{V_1,V_1}(v) = \{v\}$$

and for all domains composed from V_1 the decomposition should reflect the structure of composite state in regard of elementary states.

Example 2. For $V_1 = \{0, 1, 2\}$ and $V = \{(0,0), (0,1), (0,2), (1,0), (1,2), \ldots\}$ the $D_{V_1,V}$ is most naturally of form

$$\begin{aligned} D_{V_1,V}((0,0)) &= \{0\} \\ D_{V_1,V}((0,1)) &= \{0,1\} \\ D_{V_1,V}((0,2)) &= \{0,2\} \\ D_{V_1,V}((1,0)) &= \{0,1\} \\ D_{V_1,V}((1,2)) &= \{1,2\} \\ &\vdots \end{aligned}$$

Figure 2.2 illustrates how decomposition function may relate composite states to elementary states of its components; not all mappings are captured for better comprehensibility.

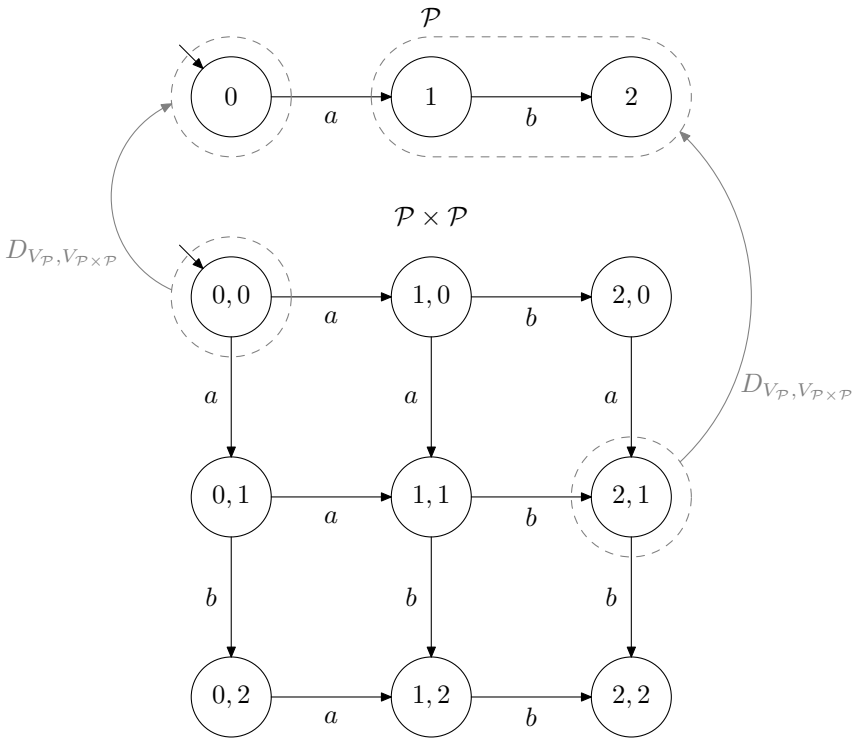


Figure 2.2: Demonstration of the meaning of state decomposition function

Depicting multiple threads requires us to consider an automaton whose states encode states of each thread and that performs scheduling of the threads by interleaving their transitions. Because of the synchronisation it is possible to consider only some states and only some transitions.

As in the case of composition even with replication there are limitations to its operands. It is not reasonable to allow to combine any two interface automata

into a multi-threaded solution.

Synchronisation itself may seem to give rise to compatibility conflicts by disallowing acceptance of a synchronised method call of a suspended thread while another thread is still holding the lock simply because there is no notion of suspension. States at beginning of critical sections in particular would often lead to illegal states in the parallelised automaton, because of a common occurance of method calls starting a critical section. In the spirit of optimistic approach it is not necessary to consider this a malicious and broken feature, but, in fact, it makes such situations distinguishable. It is possible to extract states from the set of all illegal states if the output action in question is blocked for the concrete thread by not acquiring lock beforehand. This reflexes reality in many systems where the particular method is called but waits for the lock before actually advancing into the critical section. Notice that attempting to parallelise an automaton whose initial state is critical results in an empty automaton. It is not reasonable to design such models. With this approach it is possible to define models of situations where there is no thread holding more than one lock at a time.

2.3 Unmodelled behaviour

Not all language features are going to be supported by our models. One such example are exeptions. We consider important mentioning this decision to avoid later confusion. Even though exception handling falls into a group of common mechanisms present on many platforms, it is one of those features that may make a control-flow very messy. In the realm of components it can be argued that components are designed in such a way that they do not seem to throw exceptions observable from the outside. Therefore rendering the support for modelling them unnecessary.

3. Iterative composition

The basic definitions have the deficiency of not allowing for a formal repetitive composition as all shared actions become internal and the composite automaton is inherently incomposable with any of its components which makes sense as long as there is no need for unbound transitions being added beside the already bound transitions.

This is, however, in conflict with the goals of this thesis. To be able to replicate functionality of a component it is necessary to allow coexistence of bound and unbound transitions utilising the same actions. It is possible to bind each thread of a server with one particular client separately and combine them into a final multi-threaded solution afterwards. But that does not characterise the most general type of parallel executions, although we will consider it later as well. For now it is important to consider a general approach to be able to compare the result to the most straighforward and possibly naive parallelised automata. To overcome the problem of mutually disjoint sets of actions we need to revisit the original definitions.

3.1 Relaxed interface automata

The composition of interface automata is very restrictive about which pairs of interface automata are composable. The reason is the composition and by extension the interface automata themselves.

Definition. A *relaxed interface automaton* $\mathcal{P}$ is a tuple

$$\left(V_{\mathcal{P}}, V_{\mathcal{P}}^{init}, A_{\mathcal{P}}^{I}, A_{\mathcal{P}}^{O}, A_{\mathcal{P}}^{H}, \mathcal{T}_{\mathcal{P}}^{I}, \mathcal{T}_{\mathcal{P}}^{O}, \mathcal{T}_{\mathcal{P}}^{H}\right)$$

for which

$$\begin{aligned} V_{\mathcal{P}}^{init} &\subseteq V_{\mathcal{P}} \\ |V_{\mathcal{P}}^{init}| &\leq 1 \\ \mathcal{T}_{\mathcal{P}}^{I} &\subseteq V_{\mathcal{P}} \times A_{\mathcal{P}}^{I} \times V_{\mathcal{P}} \\ \mathcal{T}_{\mathcal{P}}^{O} &\subseteq V_{\mathcal{P}} \times A_{\mathcal{P}}^{O} \times V_{\mathcal{P}} \\ \mathcal{T}_{\mathcal{P}}^{H} &\subseteq V_{\mathcal{P}} \times A_{\mathcal{P}}^{H} \times V_{\mathcal{P}} \end{aligned}$$

It should be clear that it is because we do not impose any limitations on action sets that we need to distinguish meanings of individual transitions. The reason why we still keep three sets of actions is that they are easier to use than extracting the same information from the transition sets first. And more importantly not all the actions need to be used in transitions while being part of the interface, therefore such information cannot be captured only by the transition sets.

Every interface automaton $\mathcal{P}$ corresponds to exactly one relaxed automaton $\mathcal{P}_r = \left(V_\mathcal{P}, V_\mathcal{P}^{init}, A_\mathcal{P}^I, A_\mathcal{P}^O, A_\mathcal{P}^H, \mathcal{T}_\mathcal{P}^I, \mathcal{T}_\mathcal{P}^O, \mathcal{T}_\mathcal{P}^H\right)$.

$$\begin{aligned}
\mathcal{T}_\mathcal{P}^I &= \left\{(u,a,v) \in \mathcal{T}_\mathcal{P} \middle| a \in A_\mathcal{P}^I\right\} \\
\mathcal{T}_\mathcal{P}^O &= \left\{(u,a,v) \in \mathcal{T}_\mathcal{P} \middle| a \in A_\mathcal{P}^O\right\} \\
\mathcal{T}_\mathcal{P}^H &= \left\{(u,a,v) \in \mathcal{T}_\mathcal{P} \middle| a \in A_\mathcal{P}^H\right\}
\end{aligned}$$

If we use a standard interface automaton $\mathcal{P}$ in context where a relaxed interface automaton is expected we assume $\mathcal{P}_r$ instead.

Example 3. In a standard automaton it would not be possible to depict a system such as the one in Figure 3.1, where two transitions in different roles share a common action label.

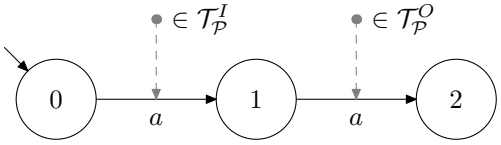


Figure 3.1: Relaxed interface automaton

3.2 Replication

Definition. *Replication function* $R_{\mathcal{P}_1}$ for a relaxed interface automaton $\mathcal{P}$, its state decomposition $D_{V_{\mathcal{P}_1},V_\mathcal{P}}$ and synchronisation annotation $N_{V_{\mathcal{P}_1}}$ denotes a function complying with the following constraints. For

$$R_{\mathcal{P}_1}\left(\mathcal{P}, D_{V_{\mathcal{P}_1},V_\mathcal{P}}, N_{V_{\mathcal{P}_1}}\right) = \left(\mathcal{Q}, D_{V_{\mathcal{P}_1},V_\mathcal{Q}}\right)$$

$$\begin{aligned}
V_\mathcal{Q} &= \left\{(v,v_1) \in V_\mathcal{P} \times V_{\mathcal{P}_1} \middle| \neg Exclude_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_\mathcal{P}}(v), \{v_1\}\right)\right\} \\
V_\mathcal{Q}^{init} &= V_\mathcal{Q} \cap \left(V_\mathcal{P}^{init} \times V_{\mathcal{P}_1}^{init}\right) \\
A_\mathcal{Q}^I &= A_\mathcal{P}^I \cup A_{\mathcal{P}_1}^I \\
A_\mathcal{Q}^O &= A_\mathcal{P}^O \cup A_{\mathcal{P}_1}^O \\
A_\mathcal{Q}^H &= A_\mathcal{P}^H \cup A_{\mathcal{P}_1}^H
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}_\mathcal{Q}^I = &\left\{((v,v_1),a,(v',v_1)) \middle| \begin{array}{l} v_1 \in V_{\mathcal{P}_1} \wedge (v,a,v') \in \mathcal{T}_\mathcal{P}^I \wedge \\ \neg Exclude_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_\mathcal{P}}(v'), \{v_1\}\right) \end{array}\right\} \cup \\
&\left\{((v,v_1),a,(v,v_1')) \middle| \begin{array}{l} v \in V_\mathcal{P} \wedge (v_1,a,v_1') \in \mathcal{T}_{\mathcal{P}_1}^I \wedge \\ \neg Exclude_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_\mathcal{P}}(v), \{v_1'\}\right) \end{array}\right\}
\end{aligned}$$

$$
\begin{aligned}
\mathcal{T}_{\mathcal{Q}}^{O} &= \left\{ \left((v, v_1), a, (v', v_1) \right) \middle| \begin{array}{l} v_1 \in V_{\mathcal{P}_1} \wedge (v, a, v') \in \mathcal{T}_{\mathcal{P}}^{O} \wedge \\ \neg Exclude_{V_{\mathcal{P}_1}} \left(D_{V_{\mathcal{P}_1}, V_{\mathcal{P}}}(v'), \{v_1\} \right) \end{array} \right\} \cup \\
&\quad \left\{ \left((v, v_1), a, (v, v_1') \right) \middle| \begin{array}{l} v \in V_{\mathcal{P}} \wedge (v_1, a, v_1') \in \mathcal{T}_{\mathcal{P}_1}^{O} \wedge \\ \neg Exclude_{V_{\mathcal{P}_1}} \left(D_{V_{\mathcal{P}_1}, V_{\mathcal{P}}}(v), \{v_1'\} \right) \end{array} \right\} \\
\mathcal{T}_{\mathcal{Q}}^{H} &= \left\{ \left((v, v_1), a, (v', v_1) \right) \middle| \begin{array}{l} v_1 \in V_{\mathcal{P}_1} \wedge (v, a, v') \in \mathcal{T}_{\mathcal{P}}^{H} \wedge \\ \neg Exclude_{V_{\mathcal{P}_1}} \left(D_{V_{\mathcal{P}_1}, V_{\mathcal{P}}}(v'), \{v_1\} \right) \end{array} \right\} \cup \\
&\quad \left\{ \left((v, v_1), a, (v, v_1') \right) \middle| \begin{array}{l} v \in V_{\mathcal{P}} \wedge (v_1, a, v_1') \in \mathcal{T}_{\mathcal{P}_1}^{H} \wedge \\ \neg Exclude_{V_{\mathcal{P}_1}} \left(D_{V_{\mathcal{P}_1}, V_{\mathcal{P}}}(v), \{v_1'\} \right) \end{array} \right\}
\end{aligned}
$$

$$
D_{V_{\mathcal{P}_1}, V_{\mathcal{Q}}} = \left\{ \left((v, v_1), D_{V_{\mathcal{P}_1}, V_{\mathcal{P}}}(v) \cup \{v_1\} \right) \middle| (v, v_1) \in V_{\mathcal{Q}} \right\}
$$

Example 4. The replication itself is a simple operation taking into account synchronisation constraints implied by critical sections as shown in Figure 2.1 with one critical section $\mathcal{C}_1$.

It should be straightforward to verify that one step of replication results in the automaton in Figure 3.2.

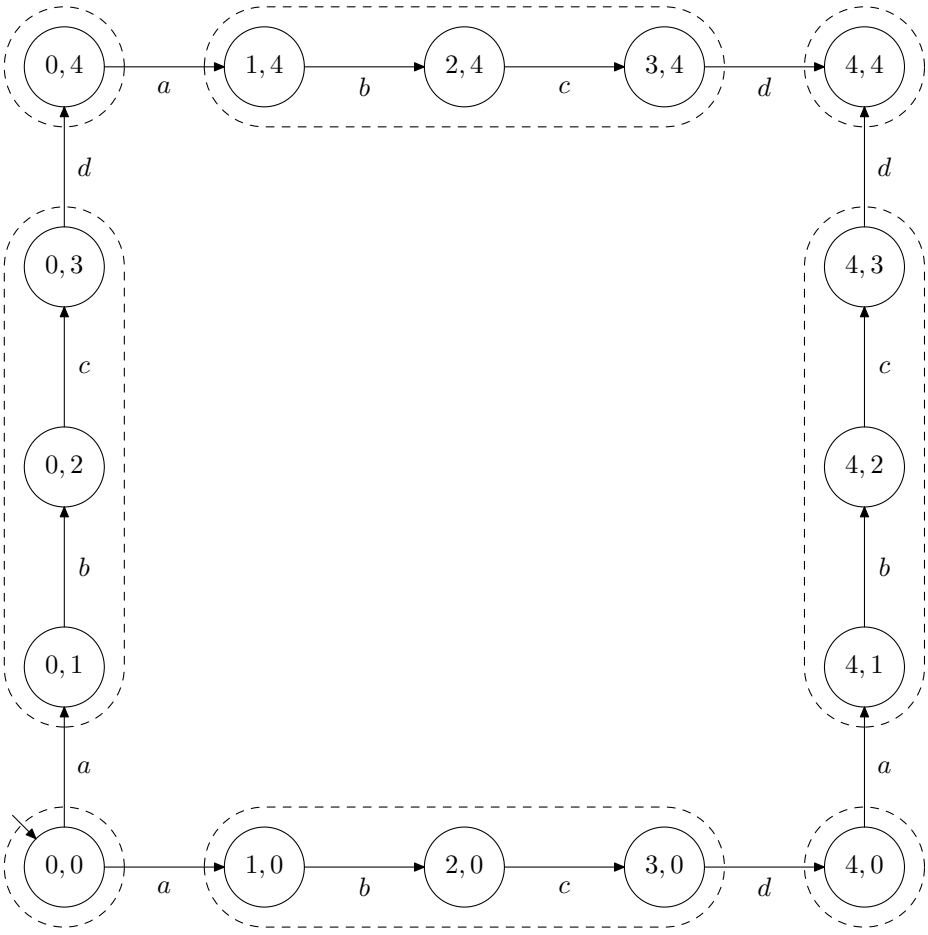


Figure 3.2: Replicated interface automaton

If we restrain to replicating one thread multiple times we can introduce a more

natural way to formulate the replication process. The following definition grants us a sequence of automata with an increasing number of threads.

Definition. Let

$$\begin{aligned}
R^0_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}\right) &= (\underbrace{\varepsilon}_{\mathcal{P}_0}, \underbrace{\{\emptyset \mapsto \emptyset\}}_{D_{V_{\mathcal{P}_1},V_{\mathcal{P}_0}}}) \\
R^1_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}\right) &= R_{\mathcal{P}_1}\left(\mathcal{P}_0, D_{V_{\mathcal{P}_1},V_{\mathcal{P}_0}}, N_{V_{\mathcal{P}_1}}\right) \\
&= \left(\mathcal{P}_1, D_{V_{\mathcal{P}_1},V_{\mathcal{P}_1}}\right) \\
R^2_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}\right) &= R_{\mathcal{P}_1}\left(\mathcal{P}_1, D_{V_{\mathcal{P}_1},V_{\mathcal{P}_1}}, N_{V_{\mathcal{P}_1}}\right) \\
&\vdots \\
R^n_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}\right) &= R_{\mathcal{P}_1}\left(\mathcal{P}_{n-1}, D_{V_{\mathcal{P}_1},V_{\mathcal{P}_{n-1}}}, N_{V_{\mathcal{P}_1}}\right)
\end{aligned}$$

The sequence of automata with an increasing degree of parallelism will be used later to compare two approaches to replication.

3.3 Composition

Before we relate the replication process to other means of parallelisation of interface automata, a composition with other automata should be considered. The main scenario in mind which led to this thesis was the ability to replicate parts of interface automata that are later composed with other automata.

It is necessary to define an adequate composition operation as we formally deviated from standard interface automata. To be able to later compare the newly defined composition with its standard preimage we will harness the possibilities of relaxed automata to postpone binding of input and output transitions of the two operands.

Definition. For interface automaton $\mathcal{P}_1$, two relaxed interface automata $\mathcal{P}$, $\mathcal{Q}$ and decomposition function $D_{V_{\mathcal{P}_1},V_{\mathcal{P}}}$ we define composition $\left(\mathcal{P}, D_{V_{\mathcal{P}_1},V_{\mathcal{P}}}\right) \otimes' \mathcal{Q} = \left(\mathcal{R}, D_{V_{\mathcal{P}_1},V_{\mathcal{R}}}\right)$ by

$$\begin{aligned}
V_{\mathcal{R}} &= V_{\mathcal{P}} \times V_{\mathcal{Q}} \\
V^{init}_{\mathcal{R}} &= V^{init}_{\mathcal{P}} \times V^{init}_{\mathcal{Q}} \\
A^I_{\mathcal{R}} &= A^I_{\mathcal{P}} \cup A^I_{\mathcal{Q}} \\
A^O_{\mathcal{R}} &= A^O_{\mathcal{P}} \cup A^O_{\mathcal{Q}} \\
A^H_{\mathcal{R}} &= A^H_{\mathcal{P}} \cup A^H_{\mathcal{Q}}
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}^I_{\mathcal{R}} = &\left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, (v'_{\mathcal{P}}, v_{\mathcal{Q}})\right) \middle| v_{\mathcal{Q}} \in V_{\mathcal{Q}} \wedge (v_{\mathcal{P}}, a, v'_{\mathcal{P}}) \in \mathcal{T}^I_{\mathcal{P}}\right\} \cup \\
&\left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, \left(v_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \middle| v_{\mathcal{P}} \in V_{\mathcal{P}} \wedge \left(v_{\mathcal{Q}}, a, v'_{\mathcal{Q}}\right) \in \mathcal{T}^I_{\mathcal{Q}}\right\}
\end{aligned}$$

$$
\begin{aligned}
\mathcal{T}^O_{\mathcal{R}} &= \left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, (v'_{\mathcal{P}}, v_{\mathcal{Q}})\right) \middle| v_{\mathcal{Q}} \in V_{\mathcal{Q}} \wedge (v_{\mathcal{P}}, a, v'_{\mathcal{P}}) \in \mathcal{T}^O_{\mathcal{P}}\right\} \cup \\
&\quad \left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, \left(v_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \middle| v_{\mathcal{P}} \in V_{\mathcal{P}} \wedge \left(v_{\mathcal{Q}}, a, v'_{\mathcal{Q}}\right) \in \mathcal{T}^O_{\mathcal{Q}}\right\} \\
\mathcal{T}^H_{\mathcal{R}} &= \left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, (v'_{\mathcal{P}}, v_{\mathcal{Q}})\right) \middle| v_{\mathcal{Q}} \in V_{\mathcal{Q}} \wedge (v_{\mathcal{P}}, a, v'_{\mathcal{P}}) \in \mathcal{T}^H_{\mathcal{P}}\right\} \cup \\
&\quad \left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, \left(v_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \middle| v_{\mathcal{P}} \in V_{\mathcal{P}} \wedge \left(v_{\mathcal{Q}}, a, v'_{\mathcal{Q}}\right) \in \mathcal{T}^H_{\mathcal{Q}}\right\} \\
D_{V_{\mathcal{P}_1}, V_{\mathcal{R}}} &= \left\{\left((v, v_r), D_{V_{\mathcal{P}_1}, V_{\mathcal{P}}}(v)\right) \middle| (v, v_r) \in V_{\mathcal{R}}\right\}
\end{aligned}
$$

Example 5. The operation unlike its standard variant basically does not recognize compatibility of the two operands. In Figure 3.3 it can be seen that the composite simulates a system where the two automata run alongside without being affected by each other.

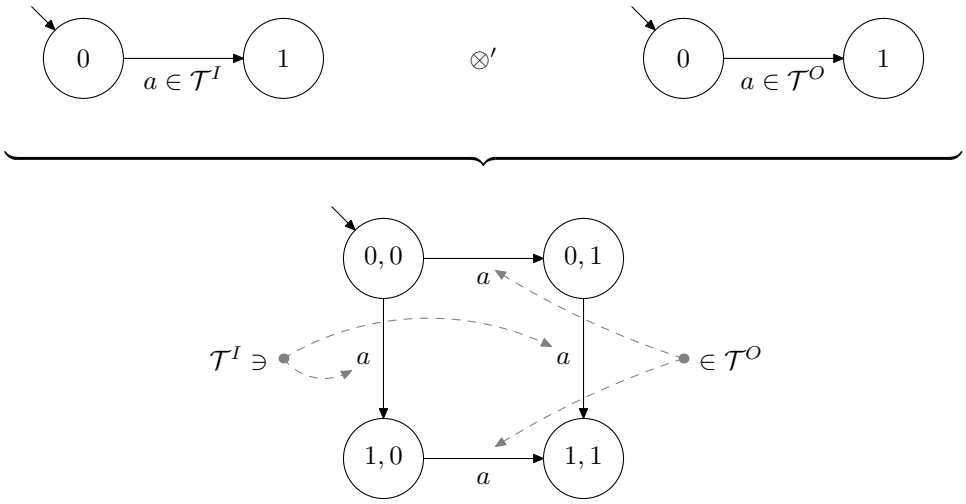


Figure 3.3: Relaxed composition

Similarly to defining a sequence of automata with increasing degree of replication, it will prove useful to denote automata with an arbitrary number of threads and composition counterparts bound to individual threads.

Definition.

$$
E_{\mathcal{P}_1}\left(\mathcal{P}, D_{V_{\mathcal{P}_1}, V_{\mathcal{P}}}, N_{\mathcal{P}_1}, \mathcal{Q}\right) = R_{\mathcal{P}_1}\left(\mathcal{P}, D_{V_{\mathcal{P}_1}, V_{\mathcal{P}}}, N_{\mathcal{P}_1}\right) \otimes' \mathcal{Q}
$$

$$
\begin{aligned}
E^0_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}, \mathcal{Q}\right) &= (\underbrace{\varepsilon}_{\mathcal{R}_0}, \underbrace{\{\emptyset \mapsto \emptyset\}}_{D_{V_{\mathcal{P}_1}, V_{\mathcal{R}_0}}}) \\
E^1_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}, \mathcal{Q}\right) &= R_{\mathcal{P}_1}\left(\mathcal{R}_0, D_{V_{\mathcal{P}_1}, V_{\mathcal{R}_0}}, N_{V_{\mathcal{P}_1}}\right) \otimes' \mathcal{Q} \\
&= \left(\mathcal{R}_1, D_{V_{\mathcal{P}_1}, V_{\mathcal{R}_1}}\right) \\
E^2_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}, \mathcal{Q}\right) &= R_{\mathcal{P}_1}\left(\mathcal{R}_1, D_{V_{\mathcal{P}_1}, V_{\mathcal{R}_1}}, N_{V_{\mathcal{P}_1}}\right) \otimes' \mathcal{Q} \\
&\vdots \\
E^n_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}, \mathcal{Q}\right) &= R_{\mathcal{P}_1}\left(\mathcal{R}_{n-1}, D_{V_{\mathcal{P}_1}, V_{\mathcal{R}_{n-1}}}, N_{V_{\mathcal{P}_1}}\right) \otimes' \mathcal{Q}
\end{aligned}
$$

3.4 Relation to standard interface automata

It is important to realise that we did not deviate from standard interface automata too much. Hence the following definition and lemma.

Definition. Let $\mathcal{P}$ be a relaxed interface automaton, then we define an operation *standardise* such that $standardise(\mathcal{P}) = \mathcal{Q}$

$$\begin{aligned}
V_{\mathcal{Q}} &= V_{\mathcal{P}} \\
V_{\mathcal{Q}}^{init} &= V_{\mathcal{P}}^{init} \\
A_{\mathcal{Q}}^{I} &= A_{\mathcal{P}}^{I} \setminus A_{\mathcal{P}}^{O} \\
A_{\mathcal{Q}}^{O} &= A_{\mathcal{P}}^{O} \setminus A_{\mathcal{P}}^{I} \\
A_{\mathcal{Q}}^{H} &= A_{\mathcal{P}}^{H} \cup \left(A_{\mathcal{P}}^{I} \cap A_{\mathcal{P}}^{O}\right) \cup \left(A_{\mathcal{P}}^{O} \cap A_{\mathcal{P}}^{I}\right) \\
\mathcal{T}_{\mathcal{Q}} &= \left\{(u,a,v) \in \mathcal{T}_{\mathcal{P}}^{I} \middle| a \notin A_{\mathcal{Q}}^{H}\right\} \cup \\
&\quad \left\{(u,a,v) \in \mathcal{T}_{\mathcal{P}}^{O} \middle| a \notin A_{\mathcal{Q}}^{H}\right\} \cup \\
&\quad \left\{(u,a,w) \middle| (u,a,v) \in \mathcal{T}_{\mathcal{P}}^{I} \wedge (v,a,w) \in \mathcal{T}_{\mathcal{P}}^{O}\right\} \cup \\
&\quad \left\{(u,a,w) \middle| (u,a,v) \in \mathcal{T}_{\mathcal{P}}^{O} \wedge (v,a,w) \in \mathcal{T}_{\mathcal{P}}^{I}\right\} \cup \\
&\quad \mathcal{T}_{\mathcal{P}}^{H}
\end{aligned}$$

The operation *standardise* can be used to convert a relaxed interface automaton to a standard interface automaton by performing binding of transitions that has been postponed or did not happen for other reasons. The following lemma explores effects of relaxed composition and subsequent standardisation.

Lemma 1. *For two composable interface automata $\mathcal{P}$, $\mathcal{Q}$, any decomposition function $D_{V_{\mathcal{P}_1},V_{\mathcal{P}}}$ and relaxed interface automaton $\mathcal{R}$ such that $\left(\mathcal{P}, D_{V_{\mathcal{P}_1},V_{\mathcal{P}}}\right) \otimes' \mathcal{Q} = \left(\mathcal{R}, D_{V_{\mathcal{P}_1},V_{\mathcal{R}}}\right)$:*

$$\mathcal{R}' = standardise(\mathcal{R}) = \mathcal{P} \otimes \mathcal{Q}$$

Proof. It is apparent from constructions in standard composition and the relaxed composition that the sets of states and initial states are the same.

If an action was shared by the two automata then it ended in both input and output sets of the composite and meanwhile no other action could have been in the two sets at the same time because it would have to be in both sets of one of the operands which cannot be true due to them being standard interface automata. Also no overlap with internal actions could have occured. Therefore for actions we get

$$\begin{aligned}
A_{\mathcal{R}'}^{I} &= \left(A_{\mathcal{P}}^{I} \cup A_{\mathcal{Q}}^{I}\right) \setminus \left(A_{\mathcal{P}}^{O} \cup A_{\mathcal{Q}}^{O}\right) \\
&= \left(A_{\mathcal{P}}^{I} \cup A_{\mathcal{Q}}^{I}\right) \setminus shared(\mathcal{P},\mathcal{Q})
\end{aligned}$$

$$
\begin{aligned}
A^O_{\mathcal{R}'} &= \left(A^O_{\mathcal{P}} \cup A^O_{\mathcal{Q}}\right) \setminus \left(A^I_{\mathcal{P}} \cup A^I_{\mathcal{Q}}\right) \\
&= \left(A^O_{\mathcal{P}} \cup A^O_{\mathcal{Q}}\right) \setminus shared(\mathcal{P}, \mathcal{Q}) \\
A^H_{\mathcal{R}'} &= A^H_{\mathcal{P}} \cup A^H_{\mathcal{Q}} \cup \left(A^I_{\mathcal{P}} \cap A^O_{\mathcal{Q}}\right) \cup \left(A^O_{\mathcal{P}} \cap A^I_{\mathcal{Q}}\right) \\
&= A^H_{\mathcal{P}} \cup A^H_{\mathcal{Q}} \cup shared(\mathcal{P}, \mathcal{Q})
\end{aligned}
$$

By analysing the rules for construction of transition sets we can verify that some conditions are sufficient for a transition to be placed in both the compared automata $\mathcal{P} \otimes \mathcal{Q}$ and $\mathcal{R}'$. Specifically

$$
\begin{aligned}
(v_{\mathcal{P}}, a, v'_{\mathcal{P}}) \in \mathcal{T}^I_{\mathcal{P}} \wedge (v_{\mathcal{Q}}, a, v_{\mathcal{Q}}) \in \mathcal{T}^O_{\mathcal{Q}} &\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v'_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \in \mathcal{T}^H_{\mathcal{P}\otimes\mathcal{Q}} \subseteq \mathcal{T}_{\mathcal{P}\otimes\mathcal{Q}} \\
&\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v'_{\mathcal{P}}, v_{\mathcal{Q}}\right)\right) \in \mathcal{T}^I_{\mathcal{R}} \wedge \\
&\quad \left(\left(v'_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v'_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \in \mathcal{T}^O_{\mathcal{R}} \\
&\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v'_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \in \mathcal{T}_{\mathcal{R}'}
\end{aligned}
$$

describing hidding of an action absorbed by the first operand after being emitted by the other, similarly for the opposite direction

$$
\begin{aligned}
(v_{\mathcal{P}}, a, v'_{\mathcal{P}}) \in \mathcal{T}^O_{\mathcal{P}} \wedge (v_{\mathcal{Q}}, a, v_{\mathcal{Q}}) \in \mathcal{T}^I_{\mathcal{Q}} &\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v'_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \in \mathcal{T}^H_{\mathcal{P}\otimes\mathcal{Q}} \subseteq \mathcal{T}_{\mathcal{P}\otimes\mathcal{Q}} \\
&\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v'_{\mathcal{P}}, v_{\mathcal{Q}}\right)\right) \in \mathcal{T}^O_{\mathcal{R}} \wedge \\
&\quad \left(\left(v'_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v'_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \in \mathcal{T}^I_{\mathcal{R}} \\
&\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v'_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \in \mathcal{T}_{\mathcal{R}'}
\end{aligned}
$$

The only other origination of an internal transition is copying the unshared transitions from the left operand

$$
\begin{aligned}
(v_{\mathcal{P}}, a, v'_{\mathcal{P}}) \in \mathcal{T}^H_{\mathcal{P}} \wedge v_{\mathcal{Q}} \in V_{\mathcal{Q}} &\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v'_{\mathcal{P}}, v_{\mathcal{Q}}\right)\right) \in \mathcal{T}^H_{\mathcal{P}\otimes\mathcal{Q}} \subseteq \mathcal{T}_{\mathcal{P}\otimes\mathcal{Q}} \\
&\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v'_{\mathcal{P}}, v_{\mathcal{Q}}\right)\right) \in \mathcal{T}^H_{\mathcal{R}} \subseteq \mathcal{T}_{\mathcal{R}'}
\end{aligned}
$$

or the right operand

$$
\begin{aligned}
\left(v_{\mathcal{Q}}, a, v'_{\mathcal{Q}}\right) \in \mathcal{T}^H_{\mathcal{Q}} \wedge v_{\mathcal{P}} \in V_{\mathcal{P}} &\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \in \mathcal{T}^H_{\mathcal{P}\otimes\mathcal{Q}} \subseteq \mathcal{T}_{\mathcal{P}\otimes\mathcal{Q}} \\
&\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \in \mathcal{T}^H_{\mathcal{R}} \subseteq \mathcal{T}_{\mathcal{R}'}
\end{aligned}
$$

Likewise it is necessary to reflect the remaining transitions that are not shared

$$
\begin{aligned}
&(v_{\mathcal{P}}, a, v'_{\mathcal{P}}) \in \mathcal{T}^I_{\mathcal{P}} \cup \mathcal{T}^O_{\mathcal{P}} \wedge \\
&a \notin shared(\mathcal{P}, \mathcal{Q}) \wedge v_{\mathcal{Q}} \in V_{\mathcal{Q}}
\end{aligned}
\quad
\begin{aligned}
&\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v'_{\mathcal{P}}, v_{\mathcal{Q}}\right)\right) \in \mathcal{T}_{\mathcal{P}\otimes\mathcal{Q}} \\
&\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v'_{\mathcal{P}}, v_{\mathcal{Q}}\right)\right) \in \mathcal{T}_{\mathcal{R}'}
\end{aligned}
$$

$$
\begin{aligned}
&\left(v_{\mathcal{Q}}, a, v'_{\mathcal{Q}}\right) \in \mathcal{T}^I_{\mathcal{Q}} \cup \mathcal{T}^O_{\mathcal{Q}} \wedge \\
&a \notin shared(\mathcal{P}, \mathcal{Q}) \wedge v_{\mathcal{P}} \in V_{\mathcal{P}}
\end{aligned}
\quad
\begin{aligned}
&\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \in \mathcal{T}_{\mathcal{P}\otimes\mathcal{Q}} \\
&\Rightarrow \left(\left(v_{\mathcal{P}}, v_{\mathcal{Q}}\right), a, \left(v_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \in \mathcal{T}_{\mathcal{R}'}
\end{aligned}
$$

If none of the above prerequisites hold for a, $v_{\mathcal{P}}$, $v'_{\mathcal{P}}$, $v_{\mathcal{Q}}$, $v'_{\mathcal{Q}}$, it can be seen that $((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, (v'_{\mathcal{P}}, v_{\mathcal{Q}}))$, $\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, \left(v_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right)$, $\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, \left(v'_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right)$ do not fall into $\mathcal{T}_{\mathcal{P}\otimes\mathcal{Q}}$ nor into $\mathcal{T}^I_{\mathcal{R}} \cup \mathcal{T}^O_{\mathcal{R}} \cup \mathcal{T}^H_{\mathcal{R}}$ which is a superset of $\mathcal{T}_{\mathcal{R}'}$ therefore the standard automata have equal sets of transitions. □

The benefit to switching to relaxed automata is that after producing a composite we can still manipulate it and modify it by replication and composition which was not possible in the same way with standard interface automata. Further more it can and will be shown that such an iterative approach produces similar results that could be obtained with standard automata provided that the number of iterations was known in advance.

Theorem 2. *For composable interface automata $\mathcal{P}_1$, $\mathcal{Q}$*

$$E^n_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}, \mathcal{Q}\right) \simeq^1 R^n_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}\right) \otimes' \underbrace{(\mathcal{Q} \times \mathcal{Q} \ldots \times \mathcal{Q})}_{\mathcal{Q}^n} = \left(R, D_{V_{\mathcal{P}_1}, V_{\mathcal{R}}}\right),$$

$$standardise(\mathcal{R}) =^2 R^n_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}\right) \otimes \mathcal{Q}^n,$$

meaning of which is that if we produce an automaton of an arbitrary degree of parallelism by the proposed operations using relaxed interface automata as an intermediate form of the model, we get an automaton that is up to the standardisation equivalent to what we could produce by naive parallelisation of a particular degree.

Proof. We will show (1) by induction. Straight from the definition the following is true

$$E^1_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}, \mathcal{Q}\right) \simeq R^1_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}\right) \otimes' \mathcal{Q}^1.$$

Now if for some i (induction hypothesis):

$$E^i_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}, \mathcal{Q}\right) = \left(\mathcal{R}_i, D_{V_{\mathcal{P}_1}, V_{\mathcal{R}_i}}\right) \simeq R^i_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}\right) \otimes' \mathcal{Q}^i = \left(\mathcal{R}'_i, D_{V_{\mathcal{P}_1}, V_{\mathcal{R}'_i}}\right)$$

we show that the same holds for $i+1$ by establishing an isomorphism.

States

Suppose a bijection between states of $\mathcal{R}_i$ and states of $\mathcal{R}'_i$

$$v_{\mathcal{R}_i} \leftrightarrow (v_{\mathcal{P}_i}, v_{\mathcal{Q}^i}) = v_{\mathcal{R}'_i}$$

it is possible to construct a bijection for states of $\mathcal{R}_{i+1}$ and $\mathcal{R}'_{i+1}$

$$((v_{\mathcal{R}_i}, v_{\mathcal{P}_1}), v_{\mathcal{Q}}) \leftrightarrow ((v_{\mathcal{P}_i}, v_{\mathcal{P}_1}), (v_{\mathcal{Q}^i}, v_{\mathcal{Q}}))$$

We know that decomposition functions from i-th step return the same sets for states of $\mathcal{R}_i$ and $\mathcal{R}'_i$ and because the composition does not affect the sets which states decompose to, we can conclude that decomposition of the $\mathcal{R}'_i$ state is determined only by $v_{\mathcal{P}_i}$.

$$\begin{gathered}
\left(\left(v_{\mathcal{R}_i}, v_{\mathcal{P}_1}\right), v_{\mathcal{Q}}\right) \in V_{\mathcal{R}_{i+1}} \\
\Updownarrow \\
\neg Exclude_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1}, V_{\mathcal{R}_i}}\left(v_{\mathcal{R}_i}\right), \{v_{\mathcal{P}_1}\}\right) \\
\Updownarrow \\
\neg Exclude_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1}, V_{\mathcal{P}_i}}\left(v_{\mathcal{P}_i}\right), \{v_{\mathcal{P}_1}\}\right) \\
\Updownarrow \\
\left(\left(v_{\mathcal{P}_i}, v_{\mathcal{P}_1}\right), \left(v_{\mathcal{Q}^i}, v_{\mathcal{Q}}\right)\right) \in V_{\mathcal{R}'_{i+1}}
\end{gathered}$$

Thanks to knowing that the decomposition functions behave the same and that composition employs full cross product during construction of the set of states, and given the partial states come from the automata in question, the presence of a state in one automaton is bound to presence of its image in the other.

Similarily a state $\left(\left(v_{\mathcal{R}_i}, v_{\mathcal{P}_1}\right), v_{\mathcal{Q}}\right)$ of $\mathcal{R}_{i+1}$ is initial if and only if all $v_{\mathcal{R}_i}$, $v_{\mathcal{P}_1}$ and $v_{\mathcal{Q}}$ are initial. The state $v_{\mathcal{R}_i}$ is initial if and only if $(v_{\mathcal{P}_i}, v_{\mathcal{Q}^i})$ is initial and then in turn both the $v_{\mathcal{P}_i}$ and $v_{\mathcal{Q}^i}$ are initial. Clearly in such a setting also $(v_{\mathcal{P}_i}, v_{\mathcal{P}_1})$ and $(v_{\mathcal{Q}^i}, v_{\mathcal{Q}})$ are initial. That is the necessary and sufficient condition for $v_{\mathcal{R}'_{i+1}}$ to be initial.

$$\left(\left(v_{\mathcal{R}_i}, v_{\mathcal{P}_1}\right), v_{\mathcal{Q}}\right) \text{is initial} \Leftrightarrow \left(\left(v_{\mathcal{P}_i}, v_{\mathcal{P}_1}\right), \left(v_{\mathcal{Q}^i}, v_{\mathcal{Q}}\right)\right) \text{is initial}$$

Actions

Because the sets of actions grow in regard to inclusion monotonically $A^{\bullet}_{\mathcal{P}_1} \subseteq A^{\bullet}_{\mathcal{R}_i}$ during replication and composition and because actions sets of powers of any interface automaton are constant

$$\begin{array}{rcccl}
A^I_{\mathcal{R}_i} & = & A^I_{\mathcal{P}_1} \cup A^I_{\mathcal{Q}} & = & A^I_{\mathcal{R}'_i} \\
A^O_{\mathcal{R}_i} & = & A^O_{\mathcal{P}_1} \cup A^O_{\mathcal{Q}} & = & A^O_{\mathcal{R}'_i} \\
A^H_{\mathcal{R}_i} & = & A^H_{\mathcal{P}_1} \cup A^H_{\mathcal{Q}} & = & A^H_{\mathcal{R}'_i}
\end{array}$$

Transitions

Once again we assume notation: $v_{\mathcal{R}_i} \leftrightarrow \left(v_{\mathcal{P}_i}, v_{\mathcal{Q}^i}\right), v'_{\mathcal{R}_i} \leftrightarrow \left(v'_{\mathcal{P}_i}, v'_{\mathcal{Q}^i}\right)$

The constraints are based on how $\mathcal{T}$'s are constructed during replication and composition.

Any transition in the composite $\mathcal{R}_{i+1}$ is of form

$$\left(\left(\left(v_{\mathcal{R}_i}, v_{\mathcal{P}_1}\right), v_{\mathcal{Q}}\right), a, \left(\left(v'_{\mathcal{R}_i}, v'_{\mathcal{P}_1}\right), v'_{\mathcal{Q}}\right)\right) \in \mathcal{T}^{\bullet}_{\mathcal{R}_{i+1}}$$

and appears if and only if it is a transition of one of its components as it can be seen from the construction of transition sets during both operations. Thus it holds that

$$\begin{gathered}
\left(\left(v_{\mathcal{R}_i}, a, v'_{\mathcal{R}_i}\right) \in \mathcal{T}^{\bullet}_{\mathcal{R}_i} \wedge v_{\mathcal{P}_1} = v'_{\mathcal{P}_1} \wedge v_{\mathcal{Q}} = v'_{\mathcal{Q}}\right) \\
\vee \\
\left(v_{\mathcal{R}_i} = v'_{\mathcal{R}_i} \wedge \left(v_{\mathcal{P}_1}, a, v'_{\mathcal{P}_1}\right) \in \mathcal{T}^{\bullet}_{\mathcal{P}_1} \wedge v_{\mathcal{Q}} = v'_{\mathcal{Q}}\right) \\
\vee \\
\left(v_{\mathcal{R}_i} = v'_{\mathcal{R}_i} \wedge v_{\mathcal{P}_1} = v'_{\mathcal{P}_1} \wedge \left(v_{\mathcal{Q}}, a, v'_{\mathcal{Q}}\right) \in \mathcal{T}^{\bullet}_{\mathcal{Q}}\right)
\end{gathered}$$

and for each clause we construct an equivalent clause that needs to hold for the other composite, yielding

$$\begin{gathered}
\left(\left(\left(v_{\mathcal{P}_i}, v_{\mathcal{Q}^i}\right), a, \left(v'_{\mathcal{P}_i}, v_{\mathcal{Q}^i}\right)\right) \in \mathcal{T}^{\bullet}_{\mathcal{R}'_i} \wedge v_{\mathcal{P}_1} = v'_{\mathcal{P}_1} \wedge v_{\mathcal{Q}} = v'_{\mathcal{Q}}\right) \\
\vee \\
\left(\left(\left(v_{\mathcal{P}_i}, v_{\mathcal{Q}^i}\right), a, \left(v_{\mathcal{P}_i}, v'_{\mathcal{Q}^i}\right)\right) \in \mathcal{T}^{\bullet}_{\mathcal{R}'_i} \wedge v_{\mathcal{P}_1} = v'_{\mathcal{P}_1} \wedge v_{\mathcal{Q}} = v'_{\mathcal{Q}}\right) \\
\vee \\
\left(\left(v_{\mathcal{P}_i}, v_{\mathcal{Q}^i}\right) = \left(v'_{\mathcal{P}_i}, v'_{\mathcal{Q}^i}\right) \wedge \left(v_{\mathcal{P}_1}, a, v'_{\mathcal{P}_1}\right) \in \mathcal{T}^{\bullet}_{\mathcal{P}_1} \wedge v_{\mathcal{Q}} = v'_{\mathcal{Q}}\right) \\
\vee \\
\left(\left(v_{\mathcal{P}_i}, v_{\mathcal{Q}^i}\right) = \left(v'_{\mathcal{P}_i}, v'_{\mathcal{Q}^i}\right) \wedge v_{\mathcal{P}_1} = v'_{\mathcal{P}_1} \wedge \left(v_{\mathcal{Q}}, a, v'_{\mathcal{Q}}\right) \in \mathcal{T}^{\bullet}_{\mathcal{Q}}\right).
\end{gathered}$$

For the first two clauses we exploit the fact that composition produces transitions changing only a substate of one automaton at a time. Therefore we can trace the transition back into $\mathcal{P}_i$ or $\mathcal{Q}^i$. The other automaton does not change its state. All the clauses suggest that only one of $\mathcal{P}_i$, $\mathcal{Q}^i$, $\mathcal{P}_1$, $\mathcal{Q}$ performs a transition and that in turn is true only if exactly one of $\mathcal{P}_{i+1}$, $\mathcal{Q}^{i+1}$ performs a transition. Which is finally equivalent to

$$\left(\left(\left(v_{\mathcal{P}_i}, v_{\mathcal{P}_1}\right), \left(v_{\mathcal{Q}^i}, v_{\mathcal{Q}}\right)\right), a, \left(\left(v'_{\mathcal{P}_i}, v'_{\mathcal{P}_1}\right), \left(v'_{\mathcal{Q}^i}, v'_{\mathcal{Q}}\right)\right)\right) \in \mathcal{T}^{\bullet}_{\mathcal{R}'_{i+1}}.$$

Decomposition function

Regardless of how obvious this part may seem it is an important step that affects future replications and therefore is of a key value to the proof by induction. We assumed

$$D_{V_{\mathcal{P}_1}, V_{\mathcal{R}_i}}(v_{\mathcal{R}_i}) = D_{V_{\mathcal{P}_1}, V_{\mathcal{R}'_i}}\left(\left(v_{\mathcal{P}'_i}, v_{\mathcal{Q}^i}\right)\right) = D$$

and because replication extends decomposed states by the state of the new thread and composition only copies over the decomposed state of the left operand for all states that are produced in that particular step,

$$D_{V_{\mathcal{P}_1},V_{\mathcal{R}_{i+1}}}(((v_{\mathcal{R}_i},v_{\mathcal{P}_1}),v_{\mathcal{Q}}))=D\cup\{v_{\mathcal{P}_1}\}$$

and because the same applies to the construction of $\mathcal{R}'_{i+1}$,

$$D_{V_{\mathcal{P}_1},V_{\mathcal{R}'_{i+1}}}(((v_{\mathcal{P}_i},v_{\mathcal{P}_1}),(v_{\mathcal{Q}^i},v_{\mathcal{Q}})))=D\cup\{v_{\mathcal{P}_1}\}$$

Hereby (1) holds.

To prove (2) we need to verify requirements of Lemma 1. If $\mathcal{P}_1$ and $\mathcal{Q}$ are composable then any $R^n_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}\right)$ is composable with $\mathcal{Q}$ and in turn with $\mathcal{Q}^n$ because replication and cross product do not alter sets of actions. The replicated automaton is a standard automaton as no ambiguity of transition actions araises. □

Corollary 1. First it is important to see that $R^n_{\mathcal{P}_1}\left(N_{V_{\mathcal{P}_1}}\right)\otimes\mathcal{Q}^n$ is an over-approximation of composition of n-threaded server with n clients. All valid states of individual threads are captured and all combinations of thread-client bindings are as well.

It is possible to operate the relaxed automata iteratively yielding the same results as if we pre-replicated server for a fixed number of clients and then composed it with all the clients at once.

What the construction uncovers is that there does not exist a strict mapping between threads and clients and that it may happen that one client communicates with different threads throughout the execution and vice versa. Although such behaviour could be permitted in some compositions and thus should be considered for generality it will be shown that it can also be avoided and that the possibility of doing so originates in the iterative approach - alternating between spawning a thread and adding a client. Depending on the concrete context and modelled situation it may or may not be assumed that such pairing takes place in the modelled situation.

By making the decision to use relaxed definition of interface automata, a possibility to separate the replication and composition is offered. Without them no partially bound intermediate results could be constructed implying need of a more complex operation. The separation itself enables us to use different policies together, a refined variants of the replication and composition operation can be used independently. Some suggestions for the refinements will be mentioned further in this thesis.

4. Ambiguity and pairing of threads

With the use of the previously defined composition there is a trade-off between generality and size of the result automaton. This fact is an immediate consequence of postponing binding of transitions which in standard composition leads to less transitions being produced in a common scenario where the two operands share actions. Although after standardisation of a relaxed automaton all unnecessary transitions are purged, the process suffers from an inevitable growth of the size of the automaton equivalent to the worst case in the standard composition. The gain of generality is, however, only a formal advantage and in some situations may be considered harmful, because of the ambiguity arising from missing distinction of individual execution paths.

In a single thread and the basic composition there either exists a binding of particular transitions or it does not but with multiple threads a naive replication and composition would often result in an existence of execution paths where two threads call a method but recieve return values addressed to the other thread, if such semantical interpretation can be made at all.

4.1 Ambiguous event delivery

In Figure 4.1 we can see a part of an automaton whose states encode substates of server thread 1, client 1, server thread 2 and client 2 in this order. Action $a\uparrow$ marks method invocation, action $a\downarrow$ then signifies return from the method, both types of transitions bear an advancement in a server and a client at one time. It can be noticed that on the highlighted path after first two steps both clients have invoked the method a but subsequently absorb the response action of the other thread. This is the result of the most general representation of possible interleavings of virtually all composed multi-threaded models.

Again such a behaviour could be considered valid in situations where we do not model method invocations in the meaning that could be considered standard. If the replies were indistinguishable and were delivered over a shared bus, all such transitions would be part of a valid model. In the rest of the models where there exist pairs of actions that are somehow related, for example method invocations and returning from them, and where migrating computation from one thread to another is not considered, it seems reasonable to disallow absorbing events produced by calls made by other threads.

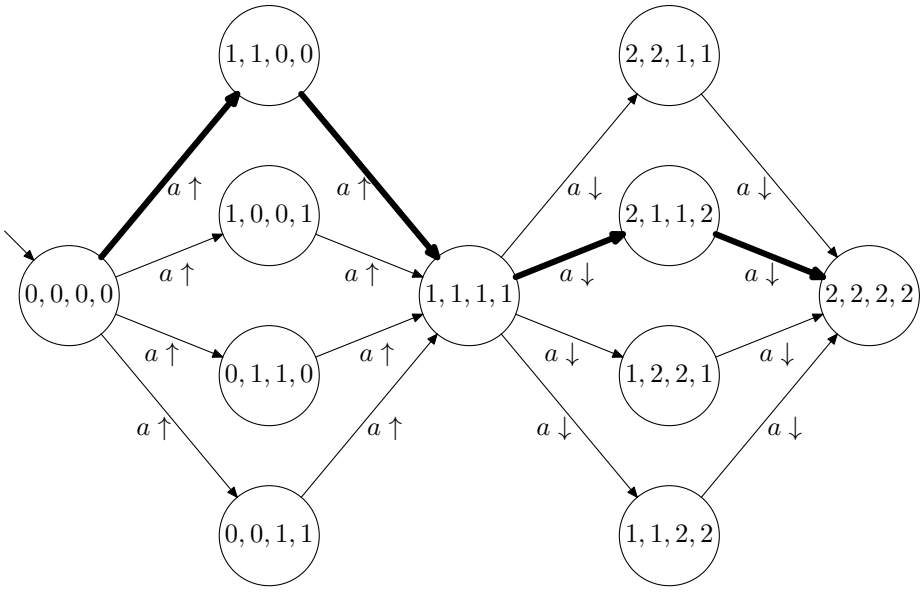


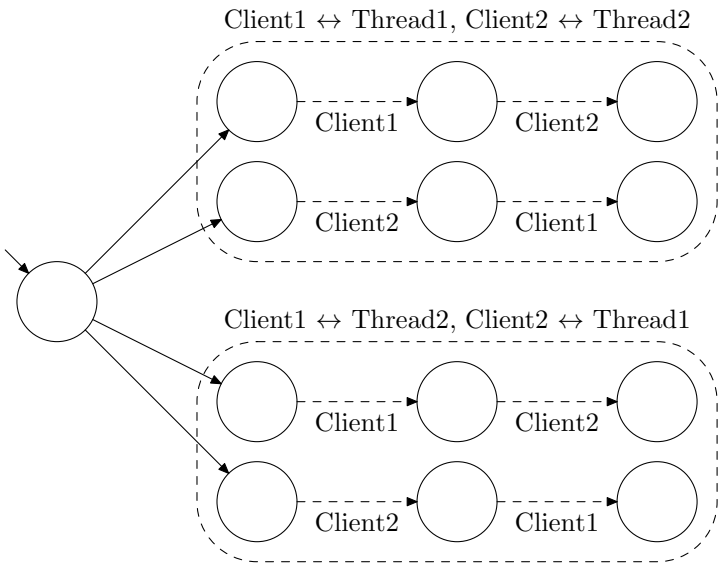
Figure 4.1: Ambiguity of composition

Pairing particular actions and maintaining affiliation of a thread to a client during execution starting with one of the actions ($a \uparrow$) and ending with the other ($a \downarrow$) would be complicated. After exploring further the structure of multi-threaded servers a more restrictive solution may be found that also resolves the ambiguity, and it is described in the following section.

4.2 Thread distinction

Even if we purged the ambiguous mixings of messages, another repercussion can be spotted in that too many structurally similar subgraphs appear in the composite. The phenomena could be interpreted in some settings as if individual server threads could be assigned to clients in any order. Such behaviour can in many cases be required, while in other cases it can be considered superfluous or even undesirable. A well-known technique to avoid such a growth of the state space is the thread canonicalisation, which is useful in situations when threads as such do not need to be identified, which happens often. Essentially a particular designation is assigned to threads and only a concrete (canonical) order of their creation or assignment is considered.

Example 6. As shown in Figure 4.2 with the previously defined composition every replicated thread gets bound with all possible counterparts resulting in a symmetrical but in many real world situations unnecessary growth of the composite automaton.



This whole branch of the composite automaton is redundant if we consider canonicalisation of threads.

Figure 4.2: Most general composition of two thread server with two clients

4.3 Pairing threads and their counterparts

Now after the similarity to standard composition has been demonstrated it is possible to refine the process for more practical uses. In fact an iterative approach allows us to save more transitions than it is possible with standard composition and possibly leads to less states being reachable and thus reduces the size of the composite automaton. We will now redefine the composition operation and consider it instead of the former one.

Naturally as it is possible to compose relaxed interface automata whose input (output) actions are already absorbing (absorbed by) transitions of another automaton, we will restrain to actions that still can be bound. In cases where we repeatedly add unbound threads there is no particular need for such a distinction but to provide a consistent tool for operating relaxed automata we will adapt *shared* to our needs. It is only the most natural adjustment that could have been anticipated since the original definition in chapter 2.

Definition. We will use $shared'(\mathcal{P}, \mathcal{R})$ for relaxed automata to denote

$$\left(A^I_{\mathcal{P}} \cap A^O_{\mathcal{R}}\right) \cup \left(A^O_{\mathcal{P}} \cap A^I_{\mathcal{R}}\right)$$

Now, inspired with the original composition, we will define a composition that allows repeated compositions which often behave better than the previous attempts to maintain generality.

Definition. For interface automaton $\mathcal{P}_1$, two relaxed interface automata $\mathcal{P}$, $\mathcal{Q}$ and decomposition function $D_{V_{\mathcal{P}_1}, V_{\mathcal{P}}}$ we define composition $\left(\mathcal{P}, D_{V_{\mathcal{P}_1}, V_{\mathcal{P}}}\right) \otimes'' \mathcal{Q} = \left(\mathcal{R}, D_{V_{\mathcal{P}_1}, V_{\mathcal{R}}}\right)$ by

$$
\begin{aligned}
V_{\mathcal{R}} &= V_{\mathcal{P}} \times V_{\mathcal{Q}} \\
V_{\mathcal{R}}^{init} &= V_{\mathcal{P}}^{init} \times V_{\mathcal{Q}}^{init} \\
A_{\mathcal{R}}^{I} &= \left(A_{\mathcal{P}}^{I} \cup A_{\mathcal{Q}}^{I}\right) \setminus shared'(\mathcal{P}, \mathcal{Q}) \\
A_{\mathcal{R}}^{O} &= \left(A_{\mathcal{P}}^{O} \cup A_{\mathcal{Q}}^{O}\right) \setminus shared'(\mathcal{P}, \mathcal{Q}) \\
A_{\mathcal{R}}^{H} &= A_{\mathcal{P}}^{H} \cup A_{\mathcal{Q}}^{H} \cup shared'(\mathcal{P}, \mathcal{Q})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}_{\mathcal{R}}^{I} = {} & \left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, (v'_{\mathcal{P}}, v_{\mathcal{Q}})\right) \middle| (v_{\mathcal{P}}, a, v'_{\mathcal{P}}) \in \mathcal{T}_{\mathcal{P}}^{I} \wedge a \notin shared'(\mathcal{P}, \mathcal{Q})\right\} \cup \\
& \left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, \left(v_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \middle| \left(v_{\mathcal{Q}}, a, v'_{\mathcal{Q}}\right) \in \mathcal{T}_{\mathcal{Q}}^{I} \wedge a \notin shared'(\mathcal{P}, \mathcal{Q})\right\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}_{\mathcal{R}}^{O} = {} & \left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, (v'_{\mathcal{P}}, v_{\mathcal{Q}})\right) \middle| (v_{\mathcal{P}}, a, v'_{\mathcal{P}}) \in \mathcal{T}_{\mathcal{P}}^{O} \wedge a \notin shared'(\mathcal{P}, \mathcal{Q})\right\} \cup \\
& \left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, \left(v_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \middle| \left(v_{\mathcal{Q}}, a, v'_{\mathcal{Q}}\right) \in \mathcal{T}_{\mathcal{Q}}^{O} \wedge a \notin shared'(\mathcal{P}, \mathcal{Q})\right\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}_{\mathcal{R}}^{H} = {} & \left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, \left(v'_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \middle| (v_{\mathcal{P}}, a, v'_{\mathcal{P}}) \in \mathcal{T}_{\mathcal{P}}^{I} \wedge \left(v_{\mathcal{Q}}, a, v'_{\mathcal{Q}}\right) \in \mathcal{T}_{\mathcal{Q}}^{O}\right\} \cup \\
& \left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, \left(v'_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \middle| (v_{\mathcal{P}}, a, v'_{\mathcal{P}}) \in \mathcal{T}_{\mathcal{P}}^{O} \wedge \left(v_{\mathcal{Q}}, a, v'_{\mathcal{Q}}\right) \in \mathcal{T}_{\mathcal{Q}}^{I}\right\} \cup \\
& \left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, (v'_{\mathcal{P}}, v_{\mathcal{Q}})\right) \middle| (v_{\mathcal{P}}, a, v'_{\mathcal{P}}) \in \mathcal{T}_{\mathcal{P}}^{H}\right\} \cup \\
& \left\{\left((v_{\mathcal{P}}, v_{\mathcal{Q}}), a, \left(v_{\mathcal{P}}, v'_{\mathcal{Q}}\right)\right) \middle| \left(v_{\mathcal{Q}}, a, v'_{\mathcal{Q}}\right) \in \mathcal{T}_{\mathcal{Q}}^{H}\right\}
\end{aligned}
$$

$$
D_{V_{\mathcal{P}_1}, V_{\mathcal{R}}} = \left\{\left((v, v_r), D_{V_{\mathcal{P}_1}, V_{\mathcal{P}}}(v)\right) \middle| (v, v_r) \in V_{\mathcal{R}}\right\}
$$

This is effectively the same as composing and standardising the result afterwards. The definition above is ment to emphasise that some overhead can be saved and that the operation is similar to the standard operation of composition. The benefit to this extension is that every replica of $\mathcal{P}_1$ (which may be viewed as a thread) is bound with one particular instance of $\mathcal{Q}$. No future composition will result in binding with the thread after it has been bound for the first time.

Moreover it is a way that demonstrates the possibility of modelling situations where all threads are equal. Casting off the distinction of threads reduces greatly the reachable state-space while essentially no practicaly viable information is lost. Partial overlap with thread canonicalisation, a common optimisation method, is another notable quality of this approach.

If we compare the results produced by repeated replication and the just defined composition with a product of a concrete construction over standard interface automata, the closest would be $(\mathcal{P} \otimes \mathcal{Q})^n$ purged of states and transitions violating mutual exclusivity of states in different threads.

5. Selective replication

The previously defined replication allows for establishing connection between relaxed automata and standard automata on a theoretical level and provides us with construction of models of multi-threaded solutions but does not offer any way to restrict multi-threading to only specific parts of the interface automaton. It is, however, possible to address such requirements by further enhancing the apparatus.

5.1 Revised definitions

First we divide sections of an automaton, as determined by synchronisation annotation, into replicable and irreplicable sections.

Definition. Let $\mathcal{R}_V$ denote a set of *replicable* sections for a synchronisation annotation $(\mathcal{S}_V, \mathcal{C}_V)$.

$$\mathcal{C}_V \subseteq \mathcal{R}_V \subseteq \left\{ [v]_{\mathcal{S}_V} \middle| v \in V \right\}$$

The set of replicable sections should be considered a part of the synchronisation annotation. Apart from the relation $Exclude$ we shall define for all $V_1 \subseteq V$

$$\begin{aligned} Synchronised_V(V_1) &\Leftrightarrow |V_1| = 1 \wedge v \in V_1 : v \notin \bigcup \mathcal{R}_V \\ Replicable_V(V_1) &\Leftrightarrow V_1 \cap \bigcup \mathcal{R}_V \neq \emptyset \end{aligned}$$

Definition. *Replication function* $R'_{\mathcal{P}_1}$ for a relaxed interface automaton $\mathcal{P}$, its state decomposition $D_{V_{\mathcal{P}_1}, V_P}$ and synchronisation annotation $N_{V_{\mathcal{P}_1}}$ denotes a function complying with the following constraints:

$$R'_{\mathcal{P}_1}\left(\mathcal{P}, D_{V_{\mathcal{P}_1}, V_\mathcal{P}}, N_{V_{\mathcal{P}_1}}\right) = \left(\mathcal{Q}, D_{V_{\mathcal{P}_1}, V_\mathcal{Q}}\right)$$

$$\begin{aligned} V_\mathcal{Q} &= \left\{ (v, v_1) \in V_\mathcal{P} \times V_{\mathcal{P}_1} \middle| \neg Exclude_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1}, V_\mathcal{P}}(v), \{v_1\}\right) \right\} \\ V_\mathcal{Q}^{init} &= V_\mathcal{Q} \cap \left(V_\mathcal{P}^{init} \times V_{\mathcal{P}_1}^{init}\right) \\ A_\mathcal{Q}^I &= A_\mathcal{P}^I \cup A_{\mathcal{P}_1}^I \\ A_\mathcal{Q}^O &= A_\mathcal{P}^O \cup A_{\mathcal{P}_1}^O \\ A_\mathcal{Q}^H &= A_\mathcal{P}^H \cup A_{\mathcal{P}_1}^H \\ D_{V_{\mathcal{P}_1}, V_\mathcal{Q}} &= \left\{ \left((v, v_1), D_{V_{\mathcal{P}_1}, V_\mathcal{P}}(v) \cup \{v_1\}\right) \middle| (v, v_1) \in V_\mathcal{Q} \right\} \end{aligned}$$

In the following constraints we will use $\bullet$ instead of $I/O/H$ to avoid the need to present all three variants. We will also break the sets into subsets to better

explain the meaning.

$$\overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} = V_{\mathcal{Q}} \times A_{\mathcal{Q}}^{\bullet} \times V_{\mathcal{Q}}$$

It is necessary to create transitions between synchronised states, states whose all threads are in the same irreplicable substate. It is important to realise that the composite states may encode substates of other automata, such information needs to be carried over from components to the composite. Hence

$$A = \left\{ ((u,v),a,(u',v)) \in \overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} \;\middle|\; \begin{array}{c} Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u,v))\right) \wedge \\ Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u',v))\right) \wedge \\ (u,a,u') \in \mathcal{T}_{\mathcal{P}}^{\bullet} \end{array} \right\}$$

and symmetrically

$$B = \left\{ ((u,v),a,(u,v')) \in \overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} \;\middle|\; \begin{array}{c} Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u,v))\right) \wedge \\ Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u,v'))\right) \wedge \\ (v,a,v') \in \mathcal{T}_{\mathcal{P}_1}^{\bullet} \end{array} \right\}$$

lastly

$$C = \left\{ ((u,v),a,(u',v')) \in \overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} \;\middle|\; \begin{array}{c} Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u,v))\right) \wedge \\ Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u',v'))\right) \wedge \\ (u,a,u') \in \mathcal{T}_{\mathcal{P}}^{\bullet} \wedge (v,a,v') \in \mathcal{T}_{\mathcal{P}_1}^{\bullet} \end{array} \right\}.$$

Next it is necessary to be able to start individual threads - old threads in the old automaton

$$D = \left\{ ((u,v),a,(u',v)) \in \overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} \;\middle|\; \begin{array}{c} Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u,v))\right) \wedge \\ Replicable_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{P}}}(u')\right) \wedge \\ (u,a,u') \in \mathcal{T}_{\mathcal{P}}^{\bullet} \end{array} \right\}$$

and the new thread

$$E = \left\{ ((u,v),a,(u,v')) \in \overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} \;\middle|\; \begin{array}{c} Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u,v))\right) \wedge \\ Replicable_{V_{\mathcal{P}_1}}(\{v'\}) \wedge \\ (v,a,v') \in \mathcal{T}_{\mathcal{P}_1}^{\bullet} \end{array} \right\}.$$

Another important point in the execution is the place where threads join with the main thread.

$$F = \left\{ ((u,v),a,(u',v)) \in \overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} \;\middle|\; \begin{array}{c} Replicable_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{P}}}(u)\right) \wedge \\ Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u',v))\right) \wedge \\ (u,a,u') \in \mathcal{T}_{\mathcal{P}}^{\bullet} \end{array} \right\}$$

$$G = \left\{ ((u,v),a,(u,v')) \in \overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} \;\middle|\; \begin{array}{c} Replicable_{V_{\mathcal{P}_1}}(\{v\}) \wedge \\ Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u,v'))\right) \wedge \\ (v,a,v') \in \mathcal{T}_{\mathcal{P}_1}^{\bullet} \end{array} \right\}$$

Only execution and scheduling of the replicated parts remain.

$$H = \left\{ ((u,v),a,(u',v)) \in \overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} \;\middle|\; \begin{array}{c} \neg Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{P}}}(u)\right) \wedge \\ \neg Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u',v))\right) \wedge \\ (u,a,u') \in \mathcal{T}_{\mathcal{P}}^{\bullet} \end{array} \right\}$$

$$I = \left\{ ((u,v),a,(u',v)) \in \overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} \;\middle|\; \begin{array}{c} \neg Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u,v))\right) \wedge \\ \neg Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{P}}}(u')\right) \wedge \\ (u,a,u') \in \mathcal{T}_{\mathcal{P}}^{\bullet} \end{array} \right\}$$

$$J = \left\{ ((u,v),a,(u',v)) \in \overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} \;\middle|\; \begin{array}{c} \neg Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u,v))\right) \wedge \\ \neg Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u',v))\right) \wedge \\ Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{P}}}(u)\right) \wedge \\ Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{P}}}(u')\right) \wedge \\ D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}(u) = D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}(u') \wedge \\ (u,a,u') \in \mathcal{T}_{\mathcal{P}}^{\bullet} \end{array} \right\}$$

$$K = \left\{ ((u,v),a,(u,v')) \in \overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} \;\middle|\; \begin{array}{c} \neg Synchronised_{V_{\mathcal{P}_1}}(\{v\}) \wedge \\ \neg Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u,v'))\right) \wedge \\ (v,a,v') \in \mathcal{T}_{\mathcal{P}_1}^{\bullet} \end{array} \right\}$$

$$L = \left\{ ((u,v),a,(u,v')) \in \overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} \;\middle|\; \begin{array}{c} \neg Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u,v))\right) \wedge \\ \neg Synchronised_{V_{\mathcal{P}_1}}(\{v'\}) \wedge \\ (v,a,v') \in \mathcal{T}_{\mathcal{P}_1}^{\bullet} \end{array} \right\}$$

$$M = \left\{ ((u,v),a,(u,v')) \in \overline{\mathcal{T}_{\mathcal{Q}}^{\bullet}} \;\middle|\; \begin{array}{c} \neg Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u,v))\right) \wedge \\ \neg Synchronised_{V_{\mathcal{P}_1}}\left(D_{V_{\mathcal{P}_1},V_{\mathcal{Q}}}((u,v'))\right) \wedge \\ Synchronised_{V_{\mathcal{P}_1}}(\{v\}) \wedge \\ Synchronised_{V_{\mathcal{P}_1}}(\{v'\}) \wedge \\ v = v' \wedge \\ (v,a,v') \in \mathcal{T}_{\mathcal{P}_1}^{\bullet} \end{array} \right\}$$

The following example gives a meaning to individual subsets.

Example 7. Consider the automaton in Figure 5.1; Figure 5.2 depicts the replicated automaton.

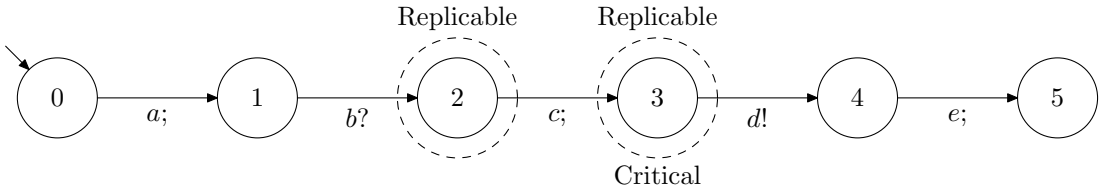


Figure 5.1: An automaton $\mathcal{P}$ with replicable section

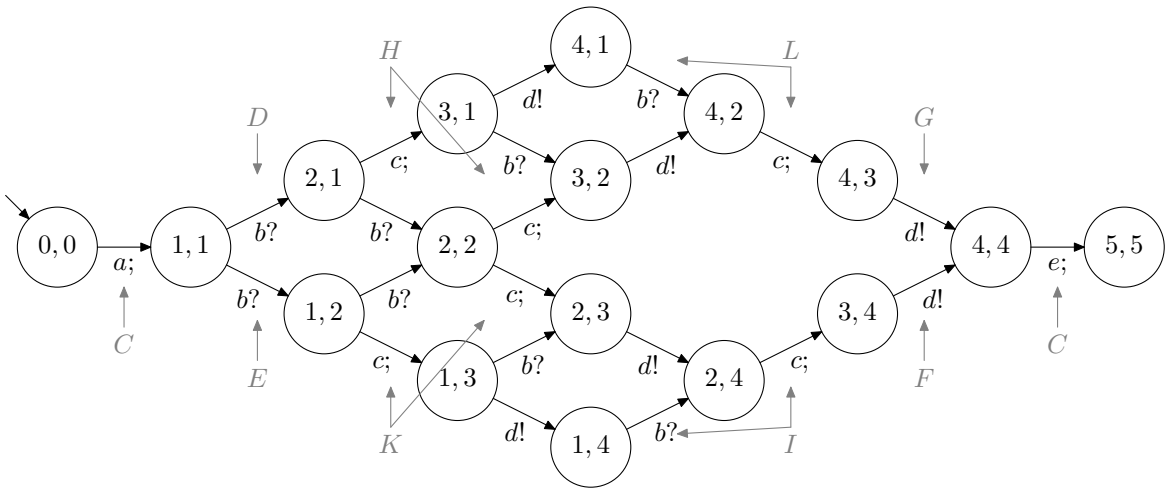


Figure 5.2: Result of a selective replication

Subsets J and M help preserve transitions performed in threads that come from a different automaton, as pictured in Figure 5.3 - the automaton $\mathcal{P}$ from Figure 5.1 is considered.

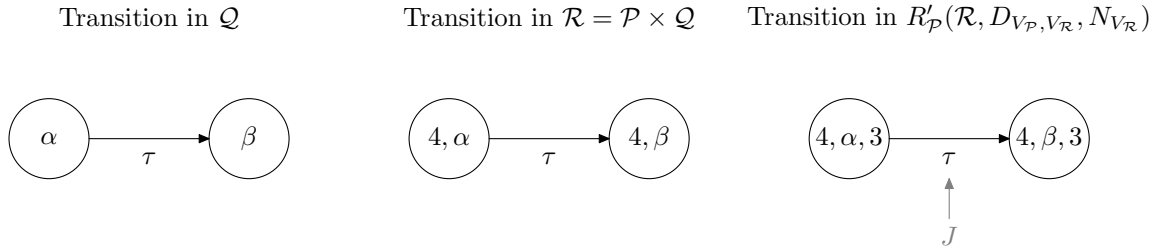


Figure 5.3: Transition of an automaton $R'_{\mathcal{P}}(\mathcal{P} \times \mathcal{Q}, \ldots)$

Subsets A, B are also distinguished in the case of automata composed of different components. For example if we consider the client in Figure 5.4.

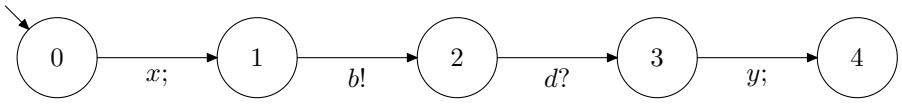


Figure 5.4: An example of client

The corresponding composition of the server and client, captured in Figure 5.5, shows interleavings of server and client transitions among states, which from the perspective of replication are not distinguished, and the transitions would disappear were there no rules as for the sets A and B.

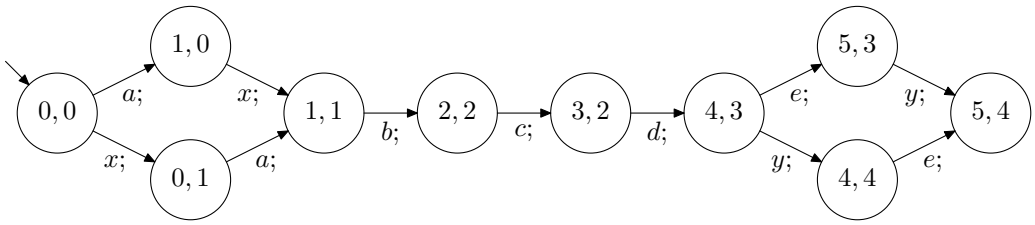


Figure 5.5: An example of single-threaded composition

Unfortunatelly an example addressing multiple threads and compositions with clients with independent and synchronised parts of state space would be too large to fit in here and even if it was not, the comprehensibility would be questionable. We believe that the principle is clear, though.

5.2 Evaluation

If we wanted to relate the operation to the previously defined one we would notice that except set C all the sets contain a transition only if it is present in one of the two automata and both its ends are valid in regard to synchronisation, which is the only necessary and in the same time also sufficient requirement for a transition to be added to a replicated automaton with the operation R. They only further restrict which transitions are meaningful in respect to replicability. If we restrained to transitions not in C the automaton produced with this newly defined replication operation would be a subautomaton of the result of the replication defined previously.

The set C merges advancement into one thread in areas of the automaton where there is no parallelism to be produced. This simulates all threads in a pool to be stopped except the main thread, which continues in a not replicated behaviour. It also can be interpreted as if the newly added thread piggybacked onto the main thread and did nothing until it reached a point where it was supposed to run and most probably listen for a request of a service.

If we imagined a possible accepted trace of action events then when ever the concrete action is handled by a not replicated section its multiplicity of allowed occurances does not change by the replication because the sets A, B and C either advance all the replicas at once or do not advance any of them, which leads to no transition being added in addition. The rest of the sets are responsible for transition between replicated and not replicated sections and for scheduling of the threaded blocks. It is important to say that we also rely on removal of unreachable parts of the automaton to keep the results meaningful. That means not all transitions are necessary for the resulting automaton but their presence is better determined by reachability of their source state.

A similar analysis of affecting the state space as we did for composition can be done for replication and although it could be argued that in the case of replicating execution branches without binding to different components (not structuraly but as instances) the branches are equivalent and redundancy appears. It should be possible to detect for a replicable section whether it needs to produce different orderings or if one canonical scheduling is sufficient to depict all states of the system. Then only an arbitrary ordering would be enabled in the resulting automaton casting off the redundancy. But optimisations such as partial order reduction can be borrowed from single-threaded automata and will not be considered here.

This type of replication also upholds the need of relaxed automata or more complex operations than standard composition, because there is not much to change about how new threads can be introduced to a standard interface automaton while keeping replication separated from composition itself.

6. Application

If we were able to assign each component an appropriate interface automaton and synchronisation annotation, it would be possible in some cases to verify whether the particular use of the component complies with the assumptions made by the automaton. Being able to extract this additional information from the implementation itself would be even better, but we shall not focus on that here. A general solution to automatisation of such a process would be hard to find as there is no universally valid parallelisation pattern across all programming languages and frameworks. Even if it existed it would probably suffer from trying to reflect completely different requirements of different platforms.

6.1 Mapping to particular implementations

The main aim is to produce a single-threaded automaton describing the behaviour of the component should no parallelism occur and only then attempt to parallelise it through replication. In the following simplified examples we leave out returns of void methods from the model in the interest of keeping the model presentable for this thesis. We also assume the interface assumptions of the following components written in Java and C++/OpenMP to be described with automaton similar to 5.1, should the ?'s, !'s and ;'s be omitted. In such a case Figure 5.2 of replicated automaton in the previous section would also apply to the two Listings and would illustrate state spaces of the multi-threaded environments listed in Listings along with the components.

In substance Listing 1 and Listing 2 implement usage of components in parallel with slight differences in the source of parallelism. In case of the server the parallelism may be described as passive as it depends on incoming requests. Notice that the bound on number of threads in Listings 1 and 2 is a part of the particular environment not the component itself. In both cases these limits are used only to demonstrate favouring of a particular scheduling of threads which is also why the sleep calls and blocking on reading from stream are used. Especially in Listing 1 it is apparent that simple patterns for parallelism can be recognised programmatically and could be used to check compatibility with the component's interface assumptions. Listing 2 captures how a naive web server could be implemented, should the bound on number of threads be put aside.

The choice of OpenMP is based on its straightforward use and ability to parallelise particular blocks. Components implemented with other technologies can indeed be modelled as well.

```
#include <omp.h>
#include <mutex>
#include <iostream>

using namespace std;

static const int N = 2;

class Component {
  private:
    mutex m;
  public:
    void a  () {cout << "a" << endl;}
    void b  (int thread) {cout << "b(" << thread << ")" << endl;}
    void cd (int thread) {
      lock_guard<mutex> l(m);
      // LOCK UNLOCK = 2 TRANSITIONS
      cout << "c(" << thread << ")" << endl;
      cout << "d(" << thread << ")" << endl;
    }
    void e  () {cout << "e" << endl;}
};

int main (int argc, char** argv) {
  omp_set_num_threads(N);

  Component c;

  c.a(); // A

  #pragma omp parallel // DO IN PARALLEL
  {
    int n = omp_get_thread_num();
    usleep((N - n - 1) * 1000);
    c.b(n); // B
    usleep(          n * 2000);
    c.cd(n); // C, D
  } // BARRIER

  c.e(); // E
}
```

Listing 1: Parallelised section in C++/OpenMP

```java
import java.io.InputStream;
import java.net.ServerSocket;

public class Server {
  void a  () {System.out.println("a");}
  void b  (int thread) {System.out.println("b(" + thread + ")");}
  synchronized void cd (int thread) {
    // LOCK UNLOCK = 2 TRANSITIONS
    System.out.println("c(" + thread + ")");
    System.out.println("d(" + thread + ")");
  }
  void e  () {System.out.println("e");}

  static int N = 2;

  public static void main(String[] args) throws Exception {
    final Server s = new Server();
    ServerSocket ss = new ServerSocket(8080);
    Thread[] connections = new Thread[N];

    s.a(); // A

    while (N > 0) { // DO IN PARALLEL
      final InputStream client = ss.accept().getInputStream();
      final int thread = --N;

      connections[N] = new Thread(new Runnable() {
        public void run() {
          try {
            client.read(); s.b(thread); // B
            client.read(); s.cd(thread); // C, D
          catch (Exception e) {}
        }
      });
      connections[N].start();
    }
    for (Thread thread : connections) { // BARRIER
      thread.join();
    }

    s.e(); // E
  }
}
```

Listing 2: TCP Server in Java

6.2 Architectures

Multi-threaded servers are not the only systems that can be modelled through replication. The parallelism, or better a particular serialisation of parallel requests, may be introduced in systems where there is only one active thread on the server side. Be it object-oriented RPC (Remote Procedure Call) systems such as CORBA [11] or non-blocking event-driven systems such as Node.js [13].

6.2.1 RPC based - CORBA

```
module example {
  interface Server {
    void a  ();
    void b  (in long client);
    void cd (in long client);
    void e  ();
  };
};
```

Listing 3: IDL of a component accessible in parallel

Suppose a component implementing an interface in Listing 3 was made accessible to multiple remote actors. Then multiple RPC requests could be delivered to the server. Even if the server was single-threaded it would process a particular interleaving of actions emitted from different clients. Any two states inside any calls would be mutually exclusive, because of single-threaded nature of processing them.

6.2.2 Event driven - Node.js

In Node.js a parallel emission of requests may result in action interleaving due to asynchronous non-blocking calls. In some situations this can be related to parallel run of multiple threads.

In Listing 4 a server is instantiated that listens for requests. Every incoming request causes a provided callback to be invoked - modelled with *request*?. The callback itself invokes a non-blocking method *read* which exits right away, which causes the target state to be an exclusive state, because all events are processed in a single thread. After returning from *read* there is a potential gap in the execution until the provided callback is fired once read data are ready. Only then a *response* is emitted. As we did earlier it would be possible to not model return

from the non-blocking asynchronous *read* but we will model it to demonstrate the little difference it makes.

```
http = require "http"
fs   = require "fs"

server = http.createServer (request, response) ->
  ###
    DO IN ‘PARALLEL‘ DUE TO ASYNC NATURE
    RESPONDS TO THE REQUEST WHOSE FILE IS READ FIRST
  ###
  fs.readFile file, (error, data) ->
    response.writeHead 200,
      "Content-Type": "text/plain"
      "Content-Length": data.length
    response.write data
    response.end()

server.listen 8080
```

Listing 4: Parallelism-like interleaving of callbacks in Coffee-Script on Node.js

Figure 6.1 further illustrates what we described above in words. The lambda transition can be replaced by rerouting *response* action back into state 0, or it could be left out completely if we did not care about endless loops.

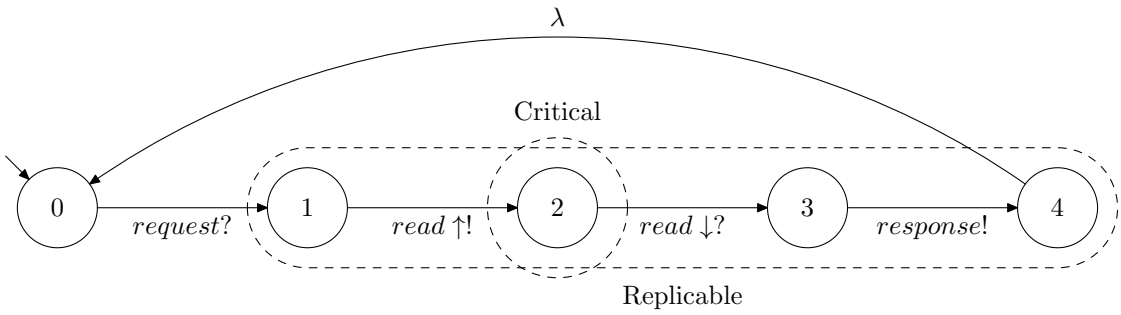


Figure 6.1: A possible model of the server implemented in Listing 4

6.3 Component systems

The previous examples described separate components with only marginal stress put on their hierarchical composition. In practice, however, composition is what motivated the definition of interface automata. While in object oriented programming languages the provided interface may and often is explicitly stated,

the required interfaces are encoded in the implementation itself and therefore they are less obvious. Component systems often employ means of definition of components on a higher level of abstraction than that of any particular programming language. The definition of provided and required interfaces is what they are ment for exactly. This encourages for extending such definitions with temporal requirements for individual invocations or message based communication. Put in other words component systems such as SOFA2 [15] or FRACTAL [12] can benefit from use of models such as interface automata. The ability to replicate automata to a particular degree of parallelism to reason about compatibility of components at the time of their composition or possibly about other types of properties verifiable by model-checking are a desirable feature of such systems if not a crucial one. The component systems form a suitable platform for the here introduced types of operations. To further demonstrate the application of the incremental replication consider the architecture depicted in Figure 6.2.

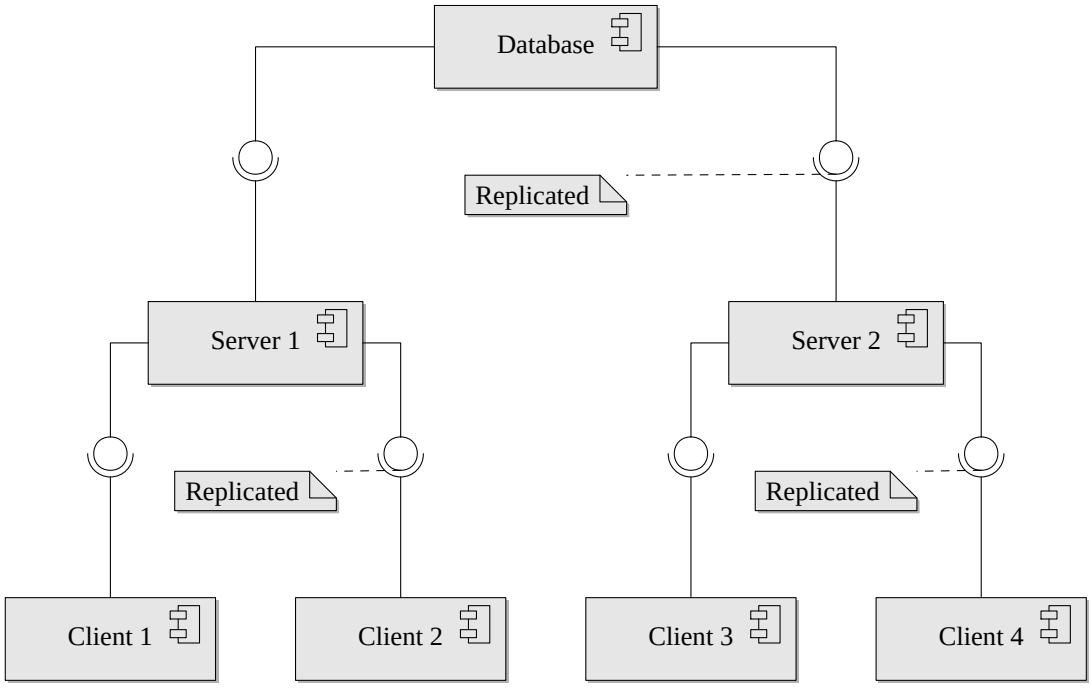


Figure 6.2: Tree-like client-server architecture encorporating replicated servers for load balancing, shared resources in form of a database and multiple isolated clients divided among the servers.

Let us specify interface automata for individual components and then provide an implementation complying with the usage protocol. We will provide only the particular implementation without a definition in ADL (Architecture Description Language) which usually preceeds implementation of business logic. Despite the fact the components in this example correspond to objects, in general this is not a necessity.

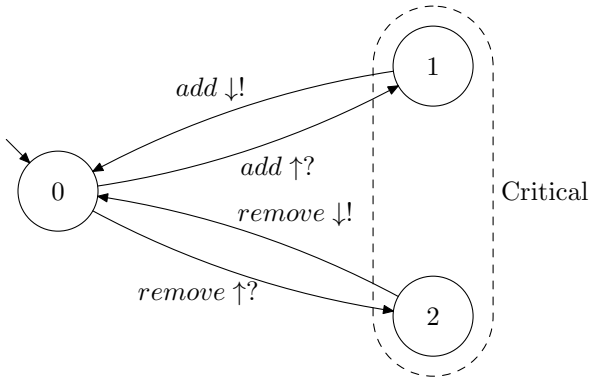


Figure 6.3: Database interface automaton

The database component provides two operations, namely *add* and *remove*. Itself it deals with atomicity of the operations which neither the servers, clients nor any hypothetical additional future components need to be aware of. Each of the two operations can be performed countless times. An implementation in C++ follows.

```
class Database {
  public:
    mutex m;
    int state;

    int add    () { lock_guard<mutex> l(m); return ++state; }
    int remove () { lock_guard<mutex> l(m); return --state; }
};
```

Listing 5: Database component written in C++

The mutual exclusivity of states where a thread can be found in its substate after initial message (method invocation) has been delivered and before a response is sent back (returning from a method) is achieved through a mutex, one of many synchronisation primitives, which is locked upon construction of lock guard whose destruction at the time of returning from the method causes the lock to be unlocked again.

The following two automata and implementations conforming to the behaviour specification describe multiple instances at a time. For our purposes both Server 1 and Server 2 are instances of the same class as well as all the clients Client 1, Client 2, Client 3 and Client4 are instances of a single class. This is a decision made purely for elimination of unnecessary obscurity of the modelled situation and to allow reuse of automata in the overall composite model.

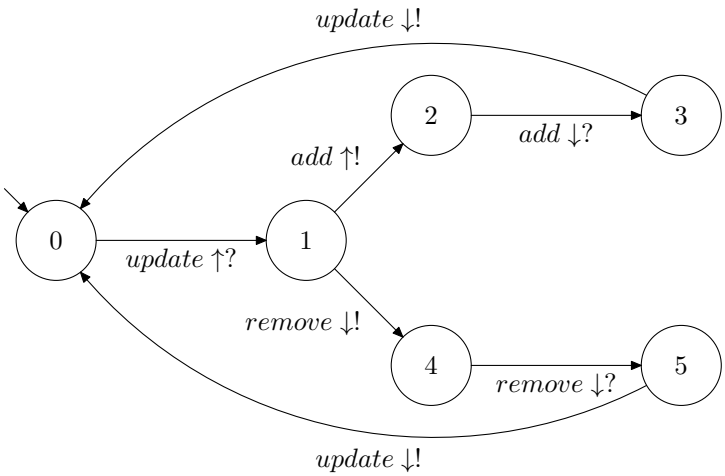


Figure 6.4: Server interface automaton

The servers implement business logic that determines interaction with database upon receiving request from a client. This is implemented by use of random choice between two branches of execution to restrain to the importance of structure of the automaton instead of going into much detail implementation-wise. Altough it can appear that the composition is also limitted and simplified, it is not the case. References to class Database could in reality employ RPC in the context of a server, which is a different component than the referred Database, to make the deployment flexible. Notice that the behaviour is not too different from what was outlined in Chapter 1. It consists of a loop that incidentally happens to split into two paths. Initialisation and termination are both void.

```
class Server {
  public:
    default_random_engine r;
    Database& database;

    Server(Database& database) : database(database), r(SEED) {}

    int update() {
      usleep(r() % 10 * 1000);

      return r() % 2 ? database.add() : database.remove();
    }
};
```

Listing 6: Server component written in C++

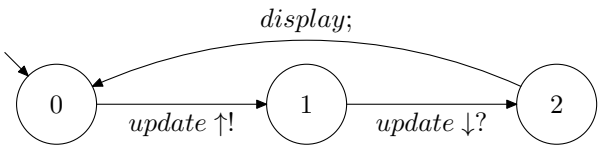


Figure 6.5: Client interface automaton

As it was the case with servers, the implementation of clients, though it may appear otherwise, is not necessarily bound to a server by possesing a reference to the particular instance living in the same address space, but rather consider utilising middleware for communicating the operation invocations to the very implementation residing possibly on a completely different node. The clients apart from communicating with servers perform local actions, represented with the inner transition *display* despite being a sequence of calls to an externally supplied object; we simply consider them local and a part of the component and we do not consider any synchronisation to take place during the mentioned transition. All functionality was placed into the constructor to avoid modelling invocations of other methods on the side of a client.

```
class Client {
  public:
    int id;
    Server& server;

    Client(Server& server, ostream& out) : server(server) {
      NEW_ID(id);

      while (true) {
        int response = server.update();

        display: out << "Client" << id << ": " << response << endl;
      }
    }
};
```

Listing 7: Client component written in C++

As it has been mentioned and can be seen from the implementation now servers and clients are unaware of the fact that their behaviour could result in blocking of a serving thread on the side of their composition counterpart, the database. This is actually a special form of a highly desirable property of composite systems, which rely on guarantees made by interfaces rather than implementation. It is important that the communication layer, message delivery middleware or method invocation, is able to enqueue messages that cannot be delivered due to the

responsible thread being blocked, resulting in an eventual message delivery or lock acquisition under a specific scheduling. In a more general setting this would become a lot more difficult once more than one lock can be held at a time by one thread.

The architecture suggests that four clients are instantiated in parallel and communicate with two distinct servers. Those in turn share a single database instance. If we wanted to construct a model of the whole system as presented above, we would compute it recursively from the individual automata as illustrated by Figure 6.6.

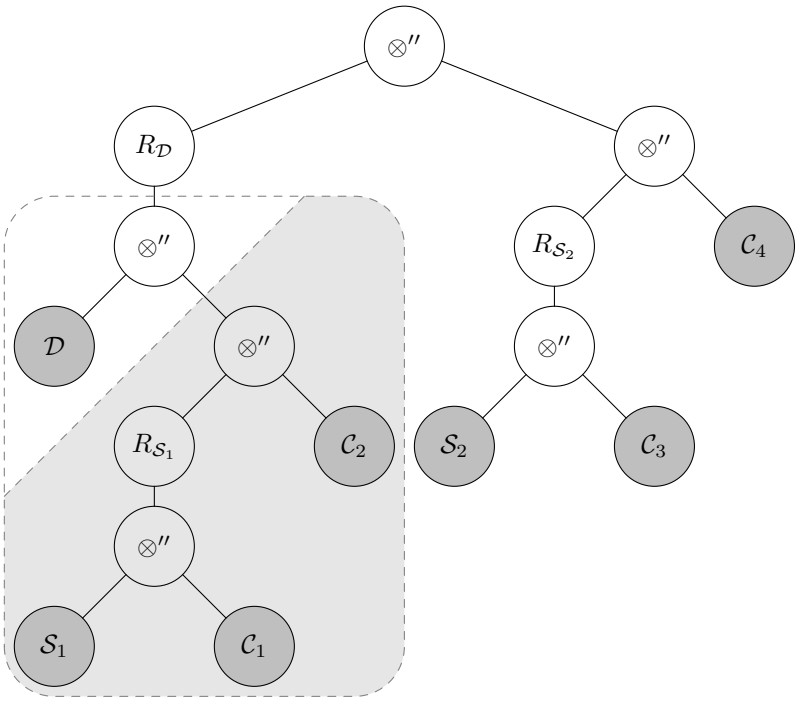


Figure 6.6: A tree of construction of model of the architecture from Figure 6.2. The leaf nodes represent input interface automata of the respective components. $\mathcal{D}$ stands for the Database component, $\mathcal{S}_i$ for the Server i and $\mathcal{C}_i$ for the Client i. The R stands for the replication function. Note that common subexpression elimination could be used provided that both servers and individual clients are all the same.

The on-demand nature of a construction of such a model can be demonstrated by considering only one server being part of the initial architecture and the additional server being added to the model by modelling its connections to clients first and then by replicating interface of the database and composing them together. The initial architecture in the mentioned settings would conform to the model inclosed by the dashed rounded rectangle. If all the servers and clients were instances of the same component definitions as it is the case in this example then the part of the initial model below the slanted dashed line can be cached and reused upon extension of the model to its desired form. This whole process

can be repeated over and over to produce models that deploy servers to more nodes for example. A different number of clients could indeed be connected to individual servers as well.

All the simplifications and decisions were made to minimise the size of the constructed automaton for the whole composite system, which is still too large to be presented here (it has thousands of states and transitions). There are ten threads in total - two database threads, two servers with two threads each and four client threads. Thanks to atomicity of database operations the database could be single threaded but in general it would not be the case. Still the upper bound $3^2 \times 6^4 \times 3^4 = 944\,784$ on the number of states of the composite automaton, which would be met for example if all states were reachable and no two states were exclusive, is almost three orders of magnitude more than the number of states in the result. Therefore it can be said that the automaton is actually quite small.

7. Related work

There have been many publications of works dealing with behaviour of not only component systems some of which address parallelism. In principal this thesis has been affected by a few of them and to various degrees. Only the relevant aspects of the related works will be highlighted.

To our best knowledge there is no similar work that would attempt to provide the same operations for interface automata. But there are other formalisms for describing functional behaviour of software systems, which also handle parallel interface bindings in component systems.

7.1 Behaviour protocols

In [3] a formalism for modelling system behaviour through expressions accepting regular languages over sets of actions has been introduced. This is where the two formalisms coincide. An interface automaton is in fact a finite state machine. Whether it is required to be deterministic or not does not make a difference. The behaviour protocols also lead to a finite state machine. They could in fact be seen as a notation for depicting such automata.

A plenty of operators are defined for protocols to be build from elementary actions or combinations of protocols. The concept of emission and absorption of messages in an atomic inner transition during composition and contravariance in required and provided interfaces during refinement is also shared between the two approaches.

The similarity can be noticed also in that the protocols also suffer from inability to encode unbounded parallelism in a particular specification. This led to further analysis and suggested solutions that follow.

7.1.1 Unbounded parallelism

A paper by Jiří Adámek [2] provides a method of dealing with systems whose models, should they be captured with existing formalisms, would become infinite due to the absence of a bound on the level of parallelism. The proposed method builds upon behaviour protocols and suggests provision of finite templates instead of models. The templates are used in particular contexts to produce models of particular levels of parallelism, which is computed recursively at the time of instantiation of the components in question.

Similarly to the solution we present this requires to move away from already established definitions in a slight way. In the case of relaxed interface automata,

however, with the approach of binding threads to the clients in each step the relaxed representation is only a temporary intermediate form and may not be considered outside the replication process. That way it can be thought of as a more complex operation applied to two standard interface automata with additional information about replicability being supplied. Therefore the procedure can be interleaved easily with other transformations and operations over interface automata.

On the other hand the solution using templates of protocols allows to parameterise the level of parallelism of a particular section with either a particular value or value computed from the architecture itself. This allows for far more complex connections to be made between individual components while maintaining distinct levels of parallelism for distinct parts of the protocol. We are convinced that restraining to parallelisation of parts of components, which we demonstrated is possible to be obtained by the proposed operations, is still useful in numerous modelled situations. What the templates cannot do is that once they are converted into instances the process cannot be repeated easily to obtain a model of the component's behaviour with a degree of parallelism larger by one.

The paper does not deal with circular dependencies between components, neither is this addressed in this thesis. On contrary it can be seen that the structure of the parse tree of the modelled situation in Figure 6.2 is similar to a graph of dependencies between components constructed during computation of maximal incoming parallelism as it is defined in the paper. The tree in Figure 6.6 captures how the final automaton is composed of the database component replicated to a degree of parallelism sufficient to serve all server-client bundles that are right operands to composition operations applied to the database.

7.2 Threaded behaviour protocols

Threaded behaviour protocols [4] are a specification language that aims at minimising the difference between the implementation and modelling languages. To do so it addresses both the notation differences and the structure of the model. It is inspired with behaviour protocols and the notation resembles Java, an example of the target implementation language of the modelled systems.

The threaded behaviour protocols separate usage of the modelled component and its own behaviour. While the behaviour of the component is described as reactions to a certain method invocations in an imperative notation, the legal usage is defined with so called provisions that capture emission of method invocations by an environment by means of extended regular expressions.

The synchronisation capabilities of the reactions of threaded behaviour pro-

tocols are more general thanks to permission of acquisition of multiple locks. Similarly to the solution from the previous section the provisions benefit from being syntactically parallelisable. It is possible to specify reentrancy, a replicability alternative in threaded behaviour protocols, already in the initial specification. Composition of two such protocols is more or less a syntactical merge of the two. The final model is constructed at the time of verification after the specification becomes closed. Then the number of threads is fixed and a particular labelled transition system with assignment can be constructed much like it is done in the solution proposed in [2]. Transitions in the model represent changes of state variables and thread stacks, by which it is possible to maintain proper pairing of method invocations and subsequent returns in one thread, which is what we also adressed in our solution. Analysis of open systems is then again based on imposing a bound on the number of threads.

While models specified with interface automata do not need to be recomputed from scratch when an incremental modification is made, the same is not true for protocol-based specifications due to the creation of a model from the complete specification and its generality. With interface automata it could be possible in theory to reuse information from a model checking of an automaton in model checking of a modified automaton. Needless to say that the solution proposed in this thesis introduces to a certain degree a reasonable alternative to the above approaches to a different formalism, interface automata.

7.3 Extended symbolic transition graphs with assignment

Unbounded parallelism is one of the reasons why models become infinite. There are different methods to cope with infinitness of models or better to remove the need for the models to be infinite completely. In a paper [6] a method of modelling infinite states within a single collapsed state is presented. Unlike in traditional labelled transition graphs such as interface automata in symbolic transition graphs the transitions contain a conditional action together with an assignment to a variable. Thanks to considering symbols, the variables, instead of particular values it is not necessary to model an infinite number of states that are being introduced due to a need to represent all the distinct values from a large domain. The extended symbolic transition graphs enhance the idea further by allowing assignments both before and after the action leading to even bigger savings.

Generally these models are at most bisimilar to the precise model and cannot be subjected to explicit-state model checking without expanding the states.

Moreover to our best knowledge there is no immediate impact on modelling of unbounded parallelism presently existing based on this method.

7.4 Parameterised contracts

While behaviour protocols tend to be less optimistic than interface automata, parametrised contracts [5] attempt to allow a component to be pronounced usable with as many environments as possible. They address a possibility of computing an interface on-demand in a specific context. If an environment does not provide all necessary functionality for a components service to function properly it is omitted and not considered provided by the component. It then cannot be used in further compositions but does not cause incompatibilities in environments that do not need the mentioned service. In a similar spirit the component can be expected to have less requirements in the case a part of the provided interface is not used in its final form.

This encourages generous provisions and their replications as present in threaded behaviour protocols with unbounded parallelisation or over-parallelisation of a given section with eventual restrictions of unused parts. The parameterised contracts where proposed mainly to address checking of interoperability. To a certain extent the idea gives us an alternative to on-demand parallelisation without restraining to a particular type of model.

Should we abandon the strict afinity to standard interface automata some of the above concepts may prove useful in pursuing a better handling of thread blocking and illegal states that result from it.

8. Future work

The current state of the apparatus still gives rise to potential extensions. Limitations arising from decisions made for a better grasp of the problem, such as restraining to isolation of individual clients, could be removed with use of more specialised composition operations. A generalised relation to parallelised automata of a particular fixed level of parallelism could be made by considering distinct interchangable clients being added into the composition in different steps of the replication process. This would be a straightforward formal enhancement of the foundations but would not directly impact the applicability.

A lot more benefit could be gained by exploring further the selective replication of individual parts of the automaton in question. The construction provided here aims to replicate only one particular part of the automaton repeatedly. It generaly does not deal with custom addition of threads in different parts of the model. The severity of such a conceptual deficiency could possibly be mitigated by provision of appropriate automata used as patterns for new threads during the replication process. Such an approach would with the highest probability require supporting changes applied to the replication operation. One possible solution could elaborate on an idea of replacing replicable sections that are not desired to be replicated in the particular step with a lambda transition from its preceeding states to its succesors bypassing the section. The replication operation would need to be modified to allow such forward jumps in threads that are effectively not being used in a particular section. Moreover equivalent branches could be drawn into the automaton and should be eliminated somehow to avoid unnecessary model growth.

An alternative path to follow would be to explore options for decomposition of an automaton and replicating individual fractions. The fractions would then need to be composed back. This approach would better compare to what can be done with behaviour protocols as mentioned above. During the analysis carried out in this thesis no particular solution taking this direction was found. In general the pursuit of an operation applicable to an automaton regardless of the history of applications of other operations that might have changed its structure makes the task harder. This is the case with application of a composition between two replications of parts of the model. The composition may copy sections, remove sections by not making a method invocation and overall obscure the structure. Even if a chain of replications produces meaningful structures, interleaving with compositions with different clients may produce varying results in which the detection of parallelisable sections may prove to be difficult.

Conclusion

It has been demonstrated that a replication of behaviour modelled by an interface automaton can be achieved with operations introduced in this thesis. That way a tool for creation of models of multi-threaded systems composed of multiple components has been provided for the formalism of interface automata. It has been shown that the operations in their most general form relate to natural expectations about the form of the resulting model but that they can be refined to better suit needs of component-based designs and that they have practical applications. Different aspects of models of threaded systems have been taken into consideration ranging from synchronisation of concurent executions to running composition avoiding confusion of message recipients and to parallelising only sections marked as replicable beforehand. Annotations of the automata to be replicated have been defined to allow features of the related components in respect to their parallel connections to be depicted by their designers to guide the replication process without any additional effort being invested at the time of composition.

Despite the fact that the replication itself utilises intermediate results whose structure is relaxed compared to interface automata, a conversion to valid interface automata is formulated and further more the conversion can be avoided completely should the replication function be used together with immediate composition. The apparatus has been defined so that it could be used repeatedly to produce models of an arbitrary level of parallelism and would merge with the already existing theory build around interface automata. To be able to benefit from a correct use of replication it is necessary however to maintain the structure of states which ultimately leads to a growth of the representation.

Particular examples have been provided to illustrate architectures that in particular can profit from the proposal. Eventually a comparison with other formal tools and their extensions has been made and advantages of different approaches have been identified. Altough a hypothetical future enhancements have been pointed out, all the goals of the thesis were fulfilled.

Bibliography

[1] Luca de Alfaro and Thomas A. Henzinger. "Interface Automata". In: *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM*. ACM Press, 2001, pp. 109–120. ISBN: 1-58113-390-1.

[2] Jiří Adámek. "Verification of Software Components: Addressing Unbounded Parallelism". In: *International Journal of Computer and Information Science* 8.2 (2007), pp. 300–309. ISSN: 1525-9293.

[3] František Plášil and Stanislav Višňovský. "Behavior Protocols for Software Components". In: *IEEE Transactions on Software Engineering* 28.11 (2002).

[4] Tomáš Poch et al. "Threaded Behavior Protocols". In: *Formal Aspects of Computing* (2011), pp. 1–30. ISSN: 0934-5043.

[5] Ralf H. Reussner. "The Use of Parameterised Contracts for Architecting Systems with Software Components". In: *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*. 2001.

[6] Weijia Deng and Huimin Lin. "Extended Symbolic Transition Graphs with Assignment". In: *Proceedings of the 29th Annual International Computer Software and Applications Conference - Volume 01*. COMPSAC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 227–232. ISBN: 0-7695-2413-3-01.

[7] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999. ISBN: 0-262-03270-8.

[8] Kim Larsen et al. "Modal I/O Automata for Interface and Product Line Theories". In: *Programming Languages and Systems*. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007. ISBN: 978-3-540-71314-2.

[9] Baier Christel and Katoen Joost-Pieter. *Principles of Model Checking*. New York: MIT Press, 2008. ISBN: 978-0-262-02649-9.

[10] Michael Emmi, Dimitra Giannakopoulou, and Corina S. Păsăreanu. *Assume-Guarantee Verification for Interface Automata*. 2008.

[11] *The Common Object Request Broker Architecture*. `http://corba.org`.

[12] *FRACTAL*. `http://fractal.ow2.org`.

[13] *Node.js*. `http://nodejs.org`.

[14] *OpenMP Application Program Interface*. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`. [Online; accessed 2013-03-15]. 2011.

[15] *SOFA2*. `http://sofa.ow2.org`.

Attachments

1. CD with an electronic copy of the thesis and copies of the key references.
2. An utility script is also provided on the CD as a proof of concept for the proposed operations. Three examples of compositions of automata whose results did not fit into the thesis are provided in form of input to the utility script as well as images of the results. Further information is present in `code/README.md` file.