TECHNISCHE UNIVERSITÄT CAROLO-WILHELMINA ZU BRAUNSCHWEIG

Diplomarbeit

Synchrony and Asynchrony in Petri Nets

Jens-Wolfhard Schicke

6. Januar 2009

Institut für Programmierung und Reaktive Systeme
Prof. Dr. Ursula Goltz

betreut durch:

Prof. Dr. Ursula Goltz
Prof. Dr. Rob van Glabbeek

Teile dieser Arbeit wurden während eines Aufenthalts am NICTA in Australien erstellt.

Abstract

In the age of multi-core processors and ubiquitous computing, more tasks than ever need to be performed by multiple, spatially disjunct computing facilities in a parallel fashion. The inherent communication delays in such systems make a purely synchronous approach infeasible. While specifying a system, assuming synchrony makes the design process simpler. It is not clear however, whether an asynchronous system can implement a synchronous specification faithfully. The present thesis gives a constructive proof that an implementation exists which is behaviourally equivalent to the specification up to a suitable linear-time equivalence. Both specification and implementation are given in Petri nets, a model well suited to describe parallelism and distribution of a system.

Keywords Asynchrony, Synchrony, Petri Nets, Distributed, Completed Step Trace Equivalence

Contents

List of Figures

1 Introduction

In today's computing world, performance depends more than ever on parallelism. As more and more systems consist of multiple processing units, software can no longer execute in a straight serial one-step-after-the-next manner if the full potential of a system needs to be realised. Rather, software must try to take as many steps in parallel as possible. While doing so, it must still behave correctly, a feat even serial software often fails to perform. Additional complexities for the parallel case emerge from an enlarged state-space and reduced debuggability due to non-determinism of scheduling.

To guide the creation of new and correct software which makes maximal use of the novel parallel technologies, mathematical models are used. These models abstract from some apparently less important aspects of the system to show particular properties about the remaining aspects. One often ignored aspect is time, in particular the duration of actions and computations. The ultimately implemented system however will be embedded in a universe which changes over time. As always when modelling, observations about the abstract model carry over into the real world only where the assumptions underlying the abstraction are valid.

There are a multitude of possibilities to abstract time based changes of the real world in a timeless model. Choosing the right abstraction for the system in question can be crucial. If too fine an abstraction is chosen, theoretical validation of the software might be infeasible, if the abstraction is too broad, the theoretically proven correctness wrt. the broad abstraction might not carry over into the real world.

To compare two different ways to abstract time, consider the example robot in Figure 1.1. It needs to enter one of the two corridors to reach its goal, a barrel of machine oil. Unfortunately, both corridors have a door, one of which will be closed. To avoid crashing into closed doors, the robot will first probe the state of the two doors before attempting to move. Drawing a diagram of the robot's mind, one might arrive at something akin to Figure 1.2. After the probing action, the robot might decide either for the left or the right door. This model however neglects the fact that the robot first decides and then moves. Making this distinction between thinking and movement explicit leads to Figure 1.3. Whether these two descriptions of the robot's mind are equivalent or not depends on which abstraction one chooses.

If one considers a world which might change arbitrarily fast, in particular faster than the robot thinks, the first model describes a robot which retains both movement options until movement has been executed, whereas the second model suggests that the robot first thinks for a while and then decides for one movement option. If the doors switched status between that decision and the attempted movement, the robot might deadlock, futilely

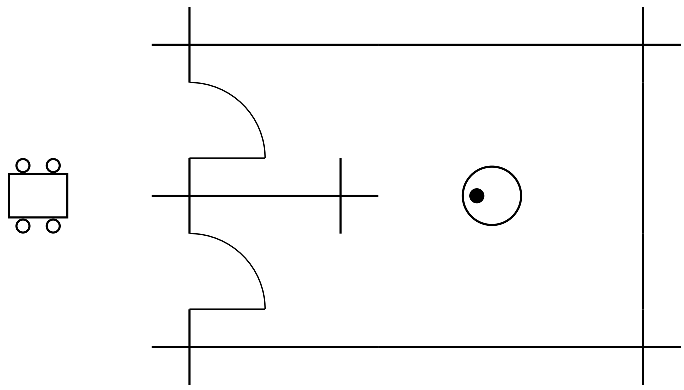

Figure 1.1: A robot wants to reach an oil barrel, yet some doors block its way

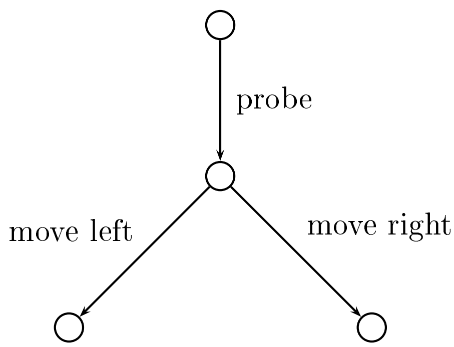

Figure 1.2: The mind of a non-thinking robot

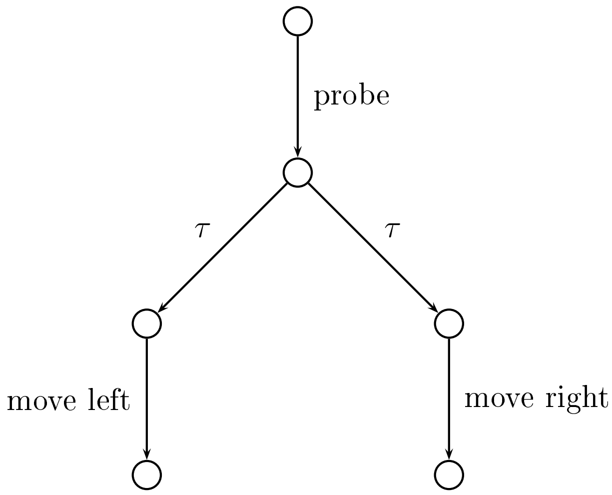

Figure 1.3: The mind of a robot which thinks for a short time

attempting to execute a now impossible movement action. Assuming a sufficiently smart robot, this difference in the outcome is only possible if the doors move faster than the robot thinks. Clearly, assuming an infinitely fast changing world is a very robust assumption. If a system can operate successfully under that assumption, it can surely operate in the real universe.

Conversely, assuming a static world sidesteps the issue of how to abstract the timing of changes therein. Under that assumption, the two models of the robot's behaviour would be considered equivalent. While such an assumption is clearly not as robust as the earlier one, today's highly integrated circuits allow the construction of robots which think substantially faster than the usual door moves. Validation of a system under the assumption that the world is static is meaningful if the computer is fast in comparison to the system it controls.

Between these two extreme assumptions, one can create a whole spectrum of different shades of time abstraction, giving rise to a spectrum of equivalence relations between behaviours. This so called linear-time branching-time spectrum has been described extensively in [4] and [6]. The frontier between linear-time and branching-time is naturally a grey area. Nonetheless, the assumption of an infinitely fast changing world corresponds to branching-time equivalences, whereas a static world assumption underlies linear-time equivalences.

The choice between different behavioural equivalences becomes even more complicated

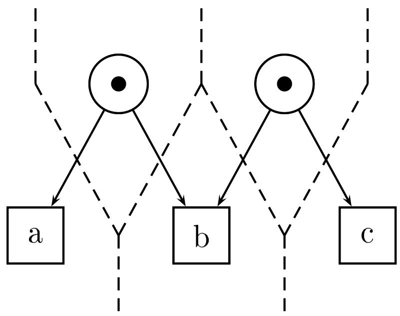

Figure 1.4: A synchronous specification and a partitioning into locations

in the light of parallelism, which is often necessary to build performant systems. One possibility is to remove parallelism by substituting it with all possible interleavings of the parallel actions, another is to allow all possible interleavings but retain a possible parallel step, yet another is to model all causal dependencies explicitly as done in pomset-trace semantics [20].

This thesis is concerned with *distributed system*, that is systems which perform activities within multiple (usually spatially) distributed locations in a coordinated fashion. Computations within different locations can naturally proceed in parallel unless they need access to shared resources. Access to these resources is often the main problem in such systems, complicated by the fact that the different locations cannot communicate instantaneously, but each message between locations must travel some distance before reaching its destination, which takes time. As no synchronous communication primitives are available such a system is called *asynchronous*.

Nonetheless, it is often easier to design a system as if synchronous communication were possible. The question then is: Given a synchronous specification of a system, can it be implemented in a distributed and hence asynchronous way? Compare Figure 1.4. A system has been specified using the synchronous model of Petri nets [19]. It has two shared resources at the top, and may either perform the actions a and c in parallel, consuming the left and right resource respectively, or it may perform b while consuming both resources at once. The elements of the system have been assigned to different locations however and can only communicate asynchronously. Is there any protocol the locations can follow in order to fulfil the synchronous specification?

The answer to that question is not a binary one. Various protocols might exist, depending on what exactly it means to “fulfil the synchronous specification”, i.e. which behavioural equivalence one uses to compare synchronous specification and distributed implementation. While it was known [7] that no protocol can exist for most branching-time equivalences, as outlined in Section 4, the question was open for linear-time equivalences.

The present thesis aims to show that, given a synchronous specification of a parallel system, a distributed implementation of this specification exists, under the assumptions that

- the environment must be slow in comparison to the implementation, i.e. the implementation is only correct up to linear-time equivalences, and
- the implementation may from time to time decide to perform steps in sequence which were parallel in the specification.

This implementation may not always be useful in the real world. If the hardware used to implement the distributed system is too slow, the real world will change faster than the system can cope with. It is my personal conjecture that a final answer about what is distributable in the real world will only be reachable by taking time fully into account. However that is out of the scope of this thesis.

The second assumption is related to the chosen concept of parallelism. This thesis assumes that whenever a system can perform two steps in parallel, these steps may also occur in sequence, which is not an unusual assumption. There is a deviation from the usual intuition however, which weens that the interleaving of events is elicited by imperfections in timing. The systems in this thesis however will decide to perform steps in explicit sequence. Some more details on this deviation are given in Section 6.

Apart from the problems about time-abstraction and parallelism considered above, there is one other problem in distributed systems which this thesis covers. Different communications between different locations in a distributed systems might proceed with different speeds. This can lead to a phenomenon called message overtaking, where messages are received in a different sequence than they were sent. This thesis makes no assumptions about properties of message overtaking at all, i.e. all forms of message overtaking are allowed.

Other problems, like content encoding within messages and error detection and recovery will be abstracted away as far as possible. Abstract interactions between parallel components are considered instead. To model these interactions and parallel components, Petri nets will be used, which allow a very intuitive and direct definition of distributability. This notion of distributability will also guarantee that no synchronous communication between different distributed components can happen.

Furthermore, as the main Petri net construction in this thesis is rather lengthy, finite state machines with a non-standard parallel combining operator will be employed as an abbreviation for a certain class of Petri nets, thus shortening the construction and the proofs.

Having now cleared up the scope of the thesis, a short overview of the contents should be next. Both Petri nets and the formal model based on state machines will be introduced in Section 2 first and then extended to a distributed setting in Section 3. Section 4 will give intuition and a short technical explanation on why certain behaviours have no distributed implementation under branching-time semantics. The main results of this thesis will be given in Section 5, where a constructive proof for a distributed implementation of Petri nets will be given. Finally Section 6 will give a conclusion and literature overview.

Proofs in the earlier chapters will only be sketched in the main text, as the results are not terribly deep and formal proofs for the Isabelle/HOL tool [17] have been created for most of them and are available in the appendix. I originally envisioned using Isabelle/HOL for the complete thesis, but abandoned that attempt after it became clear that I would not be able to complete the formal proofs within the given time frame. A short summary of the main problems encountered while working with Isabelle/HOL is given in Section 6 as well.

2 Basic Notions

As this thesis uses multisets and the notation for these is not quite standardised yet, the local version of it is given here.

Definition 2.1.1

A *multiset* M is a function which maps to natural numbers together with its domain. The domain will always stay implicit in this thesis.

An object e is an *element* of the multiset, $e \in M$, iff $M(e) > 0$.

The *union* of two multisets, $M + N$, is the pointwise addition, i.e. the multiset such that $(M + N)(e) = M(e) + N(e)$. Similarly, the *difference* of two multisets, $M - N$, is the multiset such that $(M - N)(e) = \max(M(e) - N(e), 0)$. A multiset M is a *submultiset* of another multiset N, $M \leq N$, iff $\forall x \in M.\ M(x) \leq N(x)$.

A set S can be understood within the domain of multisets by mapping all its elements to 1, i.e. $S(e) = 1 \Leftrightarrow e \in S \wedge S(e) = 0 \Leftrightarrow e \notin S$.

The *powermultiset* of a set S, $\mathcal{M}(S)$, is the set containing all multisets which only contain elements of S.

Also, the notation $\mathcal{P}(S)$ will be used to denote the powerset of a set S.

The following paragraphs about Petri nets are taken from [7], where this model has already been proven effective to describe phenomena in asynchronous systems. The main difference is that the present thesis allows transitions to carry more than one visible action. The power of this additional possibility however is only used for intermediate construction steps, and the main results hold also for nets where this is not allowed.

Definition 2.1.2

Let Act be a set of *visible actions.*

A *labelled net* N (over Act) is a tuple $(S^N, T^N, F^N, M_0^N, \ell^N)$ where

- S^N is a set (*of places*),
- T^N is a set (*of transitions*),
- $F^N \subseteq S^N \times T^N \cup T^N \times S^N$ (*the flow relation*),
- $M_0^N \subseteq S^N$ (*the initial marking*), and
- $\ell^N : T^N \to \mathcal{P}(\mathrm{Act})$ (*the labelling function*).

Petri nets are depicted by drawing the places as circles, the transitions as boxes containing the respective label, and the flow relation as arrows (*arcs*) between them. When a Petri net represents a concurrent system, a global state of such a system is given as a *marking*, a set of places, the initial state being M_0^N. A marking is depicted by placing a dot (*token*) in each of its places. The dynamic behaviour of the represented system is defined by describing the possible moves between markings. A marking M may evolve into a marking M' when a nonempty set of transitions G *fires*. In that case, for each arc $(s,t) \in F^N$ leading to a transition t in G, a token moves along that arc from s to t. Naturally, this can happen only if all these tokens are available in M in the first place. These tokens are consumed by the firing, but also new tokens are created, namely one for every outgoing arc of a transition in G. These end up in the places at the end of those arcs. A problem occurs when as a result of firing G multiple tokens end up in the same place. In that case M' would not be a marking as defined above. This thesis only considers nets in which this never happens. Such nets are called *1-safe*. Unfortunately, in order to formally define this class of nets, the firing rule must first be given without assuming 1-safety. Below this is done by forbidding the firing of sets of transitions when this might put multiple tokens in the same place.

Definition 2.1.3

Let $N = (S^N, T^N, F^N, M_0^N, \ell^N)$ be a labelled net. Let $M, M' \subseteq S^N$. The preset and postset of a net element $x \in S \cup T$ are denoted by ${}^\bullet x := \{y \mid (y,x) \in F\}$ and $x^\bullet := \{y \mid (x,y) \in F\}$ respectively. These functions are extended to sets in the usual manner, i.e. ${}^\bullet X := \{y \mid y \in {}^\bullet x, x \in X\}$.

A nonempty set of transitions $G \subseteq T^N, G \neq \emptyset$, is called a *step from M to M'*, notation $M\ [G\rangle_N\ M'$, iff

- all transitions contained in G are *enabled*, that is

$$\forall t \in G.\ {}^\bullet t \subseteq M \wedge (M \setminus {}^\bullet t) \cap t^\bullet = \emptyset\ ,$$

- all transitions of G are *independent*, that is *not conflicting*:

$$\forall t, u \in G, t \neq u.\ {}^\bullet t \cap {}^\bullet u = \emptyset \wedge t^\bullet \cap u^\bullet = \emptyset\ , \textit{and}$$

- in M' all tokens have been removed from the *preplaces* of G and new tokens have been inserted at the *postplaces* of G:

$$M' = (M \setminus {}^\bullet G) \cup G^\bullet\ .$$

To simplify statements about possible behaviours of nets, the following definition introduces some abbreviations.

Definition 2.1.4

Let $N = (S^N, T^N, F^N, M_0^N, \ell^N)$ be a labelled net. The labelling function ℓ^N shall be expanded to sets by forming the multiset union of the results, i.e. $\ell^N(G) = \sum_{t \in G} \ell^N(t)$.

- $\longrightarrow_N \subseteq \mathcal{P}(S) \times \mathcal{M}(\mathrm{Act}) \times \mathcal{P}(S)$ is given by $M \xrightarrow{A}_N M' \Leftrightarrow \exists G \subseteq T^N\!.\ M\ [G\rangle_N\ M' \wedge A = \ell^N(G) \neq \emptyset$

– $\xrightarrow{\tau}_N \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ is defined by $M \xrightarrow{\tau}_N M' \Leftrightarrow \exists t \in T.\ \ell^N(t) = \emptyset \wedge M\ [\{t\}\rangle_N\ M'$

– $\Longrightarrow_N \subseteq \mathcal{P}(S) \times \mathcal{M}(\mathrm{Act})^* \times \mathcal{P}(S)$ is defined by

$$M \xRightarrow{A_1A_2\cdots A_n}_N M' \Leftrightarrow M \xrightarrow{\tau}{}^*_N \xrightarrow{A_1}_N \xrightarrow{\tau}{}^*_N \xrightarrow{A_2}_N \cdots \xrightarrow{A_n}_N \xrightarrow{\tau}{}^*_N M'$$

where $\xrightarrow{\tau}{}^*_N$ denotes the reflexive and transitive closure of $\xrightarrow{\tau}_N$.

The following uses $M \xrightarrow{A}_N$ for $\exists M'.\ M \xrightarrow{A}_N M'$, $M \not\xrightarrow{A}_N M'$ for $\nexists M'.\ M \xrightarrow{A}_N M'$, and similar for the other two relations. Likewise $M\ [G\rangle_N$ abbreviates $\exists M'.\ M\ [G\rangle_N\ M'$.

A marking M is said to be *reachable* iff there is a $\sigma \in \mathcal{M}(\mathrm{Act})^*$ such that $M_0^N \xRightarrow{\sigma}_N M$. The set of all reachable markings is denoted by $[M_0^N\rangle$.

As stated before, only 1-safe nets are considered here. Formally, the restriction only allows *contact-free nets*, where in every reachable marking $M \in [M_0^N\rangle$ for all $t \in T$ with ${}^\bullet t \subseteq M$

$$(M \setminus {}^\bullet t) \cap t^\bullet = \emptyset\ .$$

For such nets, Definition 2.1.3 could just as well consider a transition t to be enabled in M iff ${}^\bullet t \subseteq M$, and two transitions to be independent when ${}^\bullet t \cap {}^\bullet u = \emptyset$.

Furthermore two additional restrictions are imposed. Namely that S^N and T^N are finite. Henceforth, *net* shall refer to a labelled net obeying the above restrictions.

In nets as just defined transitions are labelled with sets of *actions* drawn from a set Act. This makes it possible to see these nets as models of *reactive systems*, that interact with their environment. The firing of a transition t corresponds to the execution of the actions $\ell^N(t)$ by the system. If $\ell^N(t) \neq \emptyset$, this firing can be observed, but if $\ell^N(t) = \emptyset$, t is an *internal* or *silent* transition whose firing cannot be observed by the environment. These transitions have traditionally carried the label τ instead of $\emptyset$, and this convention will also be used in this thesis most of the time.

In the following the term *plain nets* denotes nets where ℓ^N is injective and maps only to singletons, i.e. essentially unlabelled nets. Similarly, the term *plain τ-nets* describes nets where ℓ^N maps to singletons or τ and $\ell^N(t) = \ell^N(u) \neq \tau \Rightarrow t = u$. This basically describes nets where every observable action is produced by a unique transition.

The present thesis focuses mainly on implementations of plain nets, as many of the subtleties of varying equivalence notions can thus be avoided without negatively affecting the results about asynchrony.

Some of the constructions in this thesis will lead to very large nets. Since giving them directly in Petri net notation would certainly not lead to a better understanding of the ideas guiding them, these constructions will work instead by constructing nets out of communicating finite state machines (FSMs). Since finite state machines and finite state automata are the same thing, these two terms will be used synonymously.

Definition 2.1.5

An action signature Σ is a tuple $(\Sigma_I, \Sigma_O, \Sigma_\tau)$ where

- Σ_I is a set (*of input actions*),
- Σ_O is a set (*of output actions*),
- Σ_τ is a set (*of internal actions*), and
- Σ_I, Σ_O and Σ_τ are pairwise disjoint.

In the following, Σ will also be used to mean $\Sigma_I \cup \Sigma_O \cup \Sigma_\tau$.

Definition 2.1.6

A state machine A is a tuple $(\Sigma^A, Q^A, q_0^A, \rightarrow^A)$, where

- Σ^A is an action signature,
- Q^A is a set (*of states*),
- $q_0^A \in Q^A$ (*the initial state*), and
- $\rightarrow^A \subseteq Q^A \times (\mathcal{P}(\Sigma_I^A \cup \Sigma_\tau^A) \setminus \{\emptyset\}) \times \mathcal{P}(\Sigma_O^A) \times Q^A$ (*the transition relation*).

Instead of $(q, I, O, q') \in \rightarrow^A$ the notion $q \xrightarrow{I;O}_A q'$ will be used to denote that a specific step can be performed. The state machine A is *finite*, iff Q^A is. A state $q \in Q^A$ is *reachable* iff a chain of steps $q_0^A \xrightarrow{I_1;O_1}_A \xrightarrow{I_2;O_2}_A \cdots \xrightarrow{I_n;O_n}_A q$ exists.

This definition allows systems of multiple concurrent state machines to be described as a state machine again. At the same time it allows such composed systems to perform actions in parallel, one of the main features of a truly distributed system. These features will be used in the definition of a parallel composition operator on state machines in Section 3.

Most FSMs constructed later will have the nice property of only performing one input action at a time, giving rise to the following definition.

Definition 2.1.7

A state machine A is called *serial* iff $q \xrightarrow{I;O}_A q' \Rightarrow |I| = 1$.

As the names of states of a state machine do not influence the observable behaviour of a state machine at all, it is advantageous to consider two state machines which only differ in these names as equivalent. This notion of equivalence is formalised as follows.

Definition 2.1.8

Let A and A' be two state machines.

A and A' are *isomorphic*, $A \approx A'$, if and only if $\Sigma^A = \Sigma^{A'}$ and there exists a bijection $\varphi :\ Q^A \rightarrow Q^{A'}$ such that

$$\begin{gathered} \varphi(q_0^A) = q_0^{A'} \\ q \xrightarrow{I;O}_A q' \Leftrightarrow \varphi(q) \xrightarrow{I;O}_{A'} \varphi(q')\ . \end{gathered}$$

3 Distributed Systems

As already noted, many of today's computer systems are distributed. To further analyse these systems formally, the essential aspects of distributed systems need to be singled out and converted into mathematical properties. Obviously not all of the properties should be handled in that way, otherwise the mathematical models will become convoluted and not any simpler than the original systems. Thus the formal models will be abstractions of the real systems concentrating on those aspects which seem relevant.

The formal models in this thesis will in particular ignore the possibility of hardware failures, the actual computations executed at the different locations, any knowledge about durations both of computations and of communication and any physical properties of the involved nodes like dimensions or thermal properties.

Instead the models concentrate on the possibility of parallel actions, the asynchrony of all communication between nodes and on the control flow within each of the nodes. In particular they also include the possibility of message overtaking, i.e. that two messages are received in a different order than they were sent. This phenomenon occurs not in all distributed systems, but is for example existent in the internet.

In the following, the two system models introduced in Section 2 will be extended to a distributed setting. First, nets will be equipped with a notion of locations and distribution in a pretty straightforward way, providing the intuition to connect the theoretical results to the problems of the real world. Then a parallel composition operator on state machines will be defined, producing state machines which are strongly related to distributed nets but better suited for proofs about complicated systems.

3.1 Distributed Petri Nets

To define a distributed net the easiest – and indeed obvious – way is to assume a set of locations and to mount each place and transition of the net on some element thereof. The intuition is that each element is somehow implemented at the specified location. After all elements have been placed on one location or the other, some arrows will cross location borders. It is along these arrows that the different locations communicate. An example of a net with such location information attached can be found in Figure 3.1.

A significant communication delay between locations is assumed, which can be represented explicitly by introducing τ-labelled transitions along arrows crossing location borders, as done in Figure 3.2. Note that due to this communication delay between the "start drive"

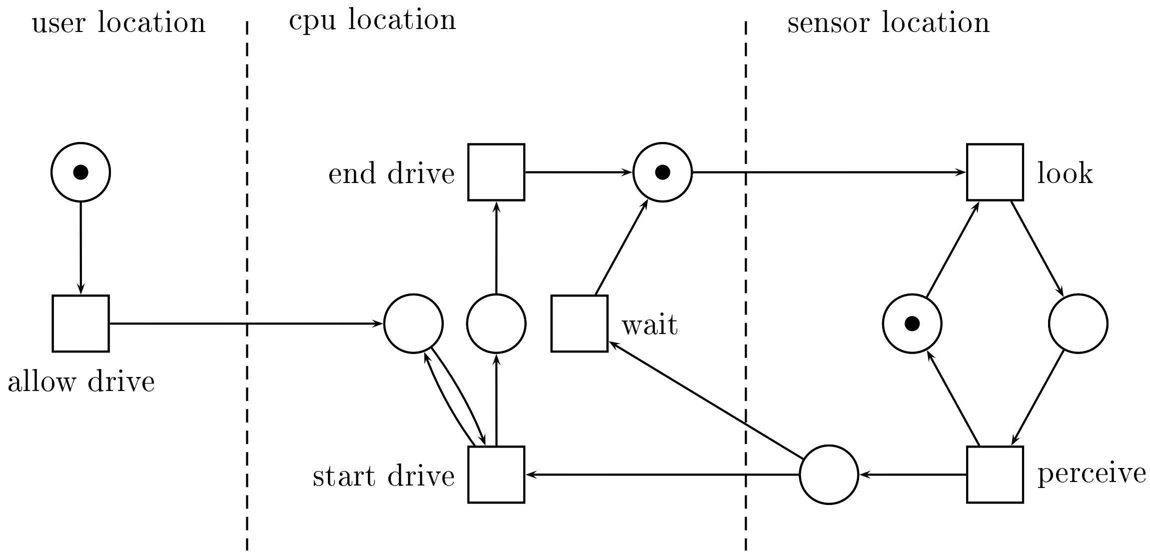

Figure 3.1: An example of a located net, modelling an example robot

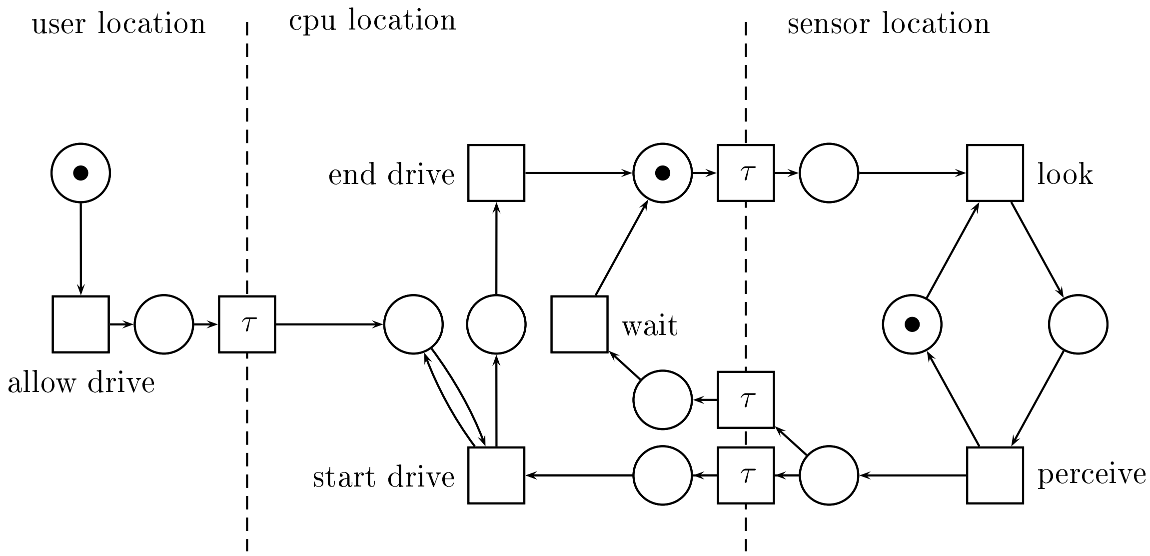

Figure 3.2: A located net with an explicit representation of communication delay

transition and its preplace to the right a premature decision is enforced, leading to a deadlock if the token is sent the wrong way and the user never allows the execution of the "allow drive" transition. A characterisation of subnets where problems of this kind are exhibited has been done in [8].

As this thesis wants to show how to implement a net in a distributed manner without changing its behaviour, making a net distributed should not introduce new deadlocks. Hence the requirement is imposed that all preplaces of a transition are co-located with the transition to enable the synchronous removal of tokens. No special requirement is necessary for connections from transitions to postplaces as all nets considered in this thesis are 1-safe. Thus the firing of transitions cannot be influenced by the presence of tokens on the postplaces. Furthermore instantaneous and delayed creation of tokens are equivalent under nearly all equivalence relations which abstract from τ-moves. Additionally, as a system is usually distributed to increase performance by using multiple execution units at the same time, a second requirement is imposed which forces transitions firing in parallel to reside on different locations.

As long as the two requirements are honoured, a system may be distributed in a variety of ways. A specific association of transitions and places to locations which fulfils these requirements is called a *valid distribution*. Some nets have multiple valid distributions, yet a single one suffices to make a net *distributed*, as it could be implemented in a distributed fashion.

Definition 3.1.1

Let $N = (S^N, T^N, F^N, M_0^N, \ell^N)$ be a net. Let Loc be a set of locations.

The net N is *distributed* iff there exists a function $\mathscr{D} : S^N \cup T^N \to \text{Loc}$ such that

- $s \in {}^\bullet t \Rightarrow \mathscr{D}(s) = \mathscr{D}(t)$ and
- $M_1 \in [M_0^N\rangle \wedge M_1\ [G\rangle_N\ M_2 \Rightarrow \forall t, u \in G, t \neq u.\ \mathscr{D}(t) \neq \mathscr{D}(u)$.

One important class of nets which are distributed are those characterized in [22] as nets of sequential machines. Sequential machines are defined therein as Petri nets with two different kinds of places. Some places are states of the sequential machine, the others are communication buffers which the machine reads and writes. As the name already suggests, sequential machines are only allowed to execute actions in sequence, not in parallel. This is formalised by partitioning the places of each sequential machine into buffer (B) and state places (S) and requiring that in each reachable marking exactly one state place holds a token. Also, to make analysing networks of sequential machines easier, one imposes that no step of a sequential machine may perform both input and output. As long as the whole network is 1-safe however, every net can be transformed into an equivalent one which fulfils this condition.

Definition 3.1.2

Let $N = (S^N, T^N, F^N, M_0^N, \ell^N)$ be a net. Let $S^N = B \cup S$ with $B \cap S = \emptyset$.

N is a *sequential machine* iff

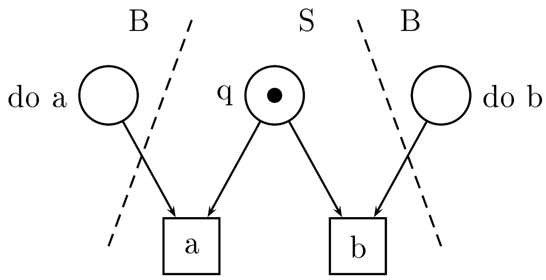

Figure 3.3: A trivial decision based upon available input – already not free choice

- $\forall t \in T^N.\ |{}^\bullet t \cap S| = 1 \wedge |t^\bullet \cap S| = 1$ (*single state invariant*) and
- $|M_0^N \cap S| = 1$ (*single state at beginning*).

The set S is called the set of *state places*, the set B the set of *buffer places*.

Actually [22] lists three other requirements. One which guarantees reachability of all states of a machine, provided enough input is available in the buffers. This was necessary as the paper tried to make all transitions in a system life. The requirement has been dropped here, as it was considered unnecessary and cumbersome, especially when dealing with initialisation sequences machines might want to perform only once. The second dropped requirement enforced the free-choice property ($\forall s \in S^N.\ |s^\bullet| > 1 \Rightarrow \forall t \in s^\bullet.\ |{}^\bullet t| = 1$) within each sequential component, effectively prohibiting sequential machines to react differently to different inputs (compare Figure 3.3). While handy to prove liveness properties, this requirement makes it impossible to transmit meaningful information to another sequential component, as the receiver can not base any decision on received input. See Section 6.2 on why this requirement is only problematic under some implicit assumptions made so far. The third dropped requirement demanded that transitions would not perform input and output at the same time. A 1-safe system however can be transformed into a semantically equivalent one which fulfils this requirement by splitting every transition in two, connected with a state place.

Sequential machines can be coupled by sharing common buffer places. To remove the necessity of locking algorithms on the lower level, each buffer is only allowed to be written by one machine and read by one machine. Hence each buffer provides a one-way communication link between a pair of machines.

Definition 3.1.3

Let $\{N_i \mid 1 \leq i \leq n\}$ with $N_i = (S^{N_i}, T^{N_i}, F^{N_i}, M_0^{N_i}, \ell^{N_i})$ be a set of sequential machines. Let S_i and B_i denote the respective state places and buffer places.

The set is *compatible* iff

- $i \neq j \Rightarrow S_i \cap S_j = \emptyset$,
- $\forall p.\ p \in B_i \wedge p \in B_j \wedge p \in B_k \Rightarrow i = j \vee j = k \vee k = i$,
- $i \neq j \Rightarrow {}^\bullet T^{N_i} \cap {}^\bullet T^{N_j} = \emptyset$, and
- $i \neq j \Rightarrow T^{N_i \bullet} \cap T^{N_j \bullet} = \emptyset$.

Definition 3.1.4
Let $\{N_i \mid 1 \leq i \leq n\}$ be a compatible set of sequential machines.

The parallel composition of the machines $N_0, N_1, \ldots, N_n$, $\big\|_{1\leq i\leq n} N_i$ is defined as the net $N_\| = (\bigcup_{1\leq i\leq n} S^{N_i}, \bigcup_{1\leq i\leq n} T^{N_i}, \bigcup_{1\leq i\leq n} F^{N_i}, \bigcup_{1\leq i\leq n} M_0^{N_i}, \bigcup_{1\leq i\leq n} \ell^{N_i})$, where the labelling function is handled as a relation.

Every network of sequential machines has a valid distribution as follows. Each sequential machine is associated with a new location to which all transitions of that sequential machine and all their preplaces belong. As the sets of preplaces of different sequential machines are guaranteed to be disjunct, such a distribution always exists.

3.2 Asynchronous Finite State Machines

It is the goal of this thesis to show how to implement arbitrary nets by distributed nets. Indeed the nets constructed will be nets of coupled sequential machines. However, the construction shown later is rather lengthy. To increase readability and understanding, the sequential machines are not represented by nets directly, but as FSMs. To ensure close correspondence between the FSMs and the nets, the coupling between FSMs is defined here rather unusually, with semantics mimicking the net behaviour.

When combining multiple FSMs into one bigger system, outputs of one machine and inputs of the other together constitute a communication link between the two machines. Such a communication link will not be observable from the outside of the composed system. All other individual actions however stay visible and constitute the outside interface of the new system. To remove the possibility of conflicts between the two machines when dealing with the outside world, all resulting input and output actions of the new system must originate uniquely from one of the two machines. To ease presentation, the additional – and semantically irrelevant – requirement is imposed that the internal actions are globally unique.

Definition 3.2.1
Two action signatures Σ and Σ' *match*, iff

- $\Sigma_I \cap \Sigma'_I = \emptyset$
- $\Sigma_O \cap \Sigma'_O = \emptyset$
- $\Sigma_\tau \cap \Sigma' = \emptyset$
- $\Sigma \cap \Sigma'_\tau = \emptyset$

To define how the composition of state machines behaves, the properties of the communication links need to be given. To avoid special cases, communication links are modelled as a queue capable of holding any amount of messages the sender might ever produce. It will turn out later, however, that all state machines actually constructed in this thesis will never send a message into a non-empty queue.

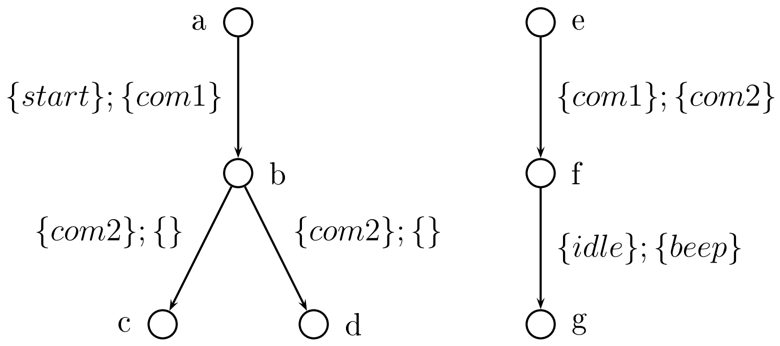

Figure 3.4: Two (serial) FSMs with matching action signatures, in particular the signature of the left FSM is $\Sigma = (\{start, com2\}, \{com1\}, \emptyset)$ and the right FSM has the signature $\Sigma' = (\{com1\}, \{com2, beep\}, \{idle\})$

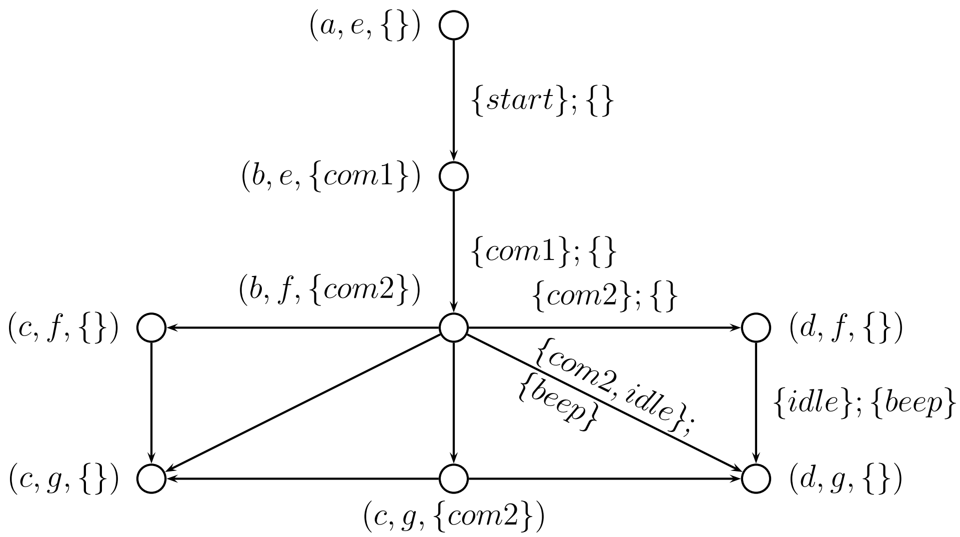

Figure 3.5: The composition of the two FSMs of Figure 3.4, again an FSM (unreachable states not shown), the signature is $\Sigma = (\{start\}, \{beep\}, \{idle, com1, com2\})$

Definition 3.2.2

Let $S = \{A_i \mid 1 \leq i \leq n\}$ be a set of state machines with pairwise matching action signatures, i.e. for all $1 \leq i \leq n, 1 \leq j \leq n$ with $i \neq j$, Σ^{A_i} and Σ^{A_j} match.

Let $I_S = \bigcup_{1 \leq i \leq n} \Sigma_I^{A_i}$, $O_S = \bigcup_{1 \leq i \leq n} \Sigma_O^{A_i}$ and $T_S = \bigcup_{1 \leq i \leq n} \Sigma_\tau^{A_i}$.

The *asynchronous parallel composition* of $A_1, A_2, \ldots, A_n$, $\big\|_{1 \leq i \leq n} A_i$, is defined as the state machine $A_\| = (\Sigma^{A_\|}, Q^{A_\|}, q_0^{A_\|}, \rightarrow^{A_\|})$ with

- $\Sigma^{A_\|} = (I_S \setminus O_S, O_S \setminus I_S, T_S \cup (I_S \cap O_S))$,
- $Q^{A_\|} = \bigtimes_{1 \leq i \leq n} Q^{A_i} \times \mathcal{M}(I_S \cap O_S)$,
- $q_0^{A_\|} = (q_0^{A_1}, \ldots, q_0^{A_n}, \emptyset)$,

and, for $I \subseteq \Sigma_I^{A_\|} \cup \Sigma_\tau^{A_\|}$ and $O \subseteq \Sigma_O^{A_\|}$, $(q_1, \ldots, q_n, M) \xrightarrow{I;O}_{A_\|} (q_1', \ldots, q_n', M')$ if and only if

- for all $1 \leq i \leq n$ either $p_i \xrightarrow{I_i;O_i}_{A_i} q_i \wedge I_i \cap \Sigma_I^{A_i} \cap \Sigma_\tau^{A_\|} \subseteq M$ or $I_i = O_i = \emptyset \wedge p_i = q_i$,
- $I = \bigcup_{1 \leq i \leq n} I_i \neq \emptyset$ (*input is composed of subcomponent inputs*),
- $O = \bigcup_{1 \leq i \leq n} O_i \cap \Sigma_O^{A_\|}$ (*output is composed of visible subcomponent outputs*), and
- $M' = (M - I) + (\bigcup_{1 \leq i \leq n} O_i \cap \Sigma_\tau^{A_\|})$ (*message buffer is correctly adjusted*).

Section 6.2 contains a discussion of the differences between this definition of state machine composition other definitions found in the literature.

Using a multiset for the message buffering requires potentially unbounded storage. However, this facility will not be used in the main construction of this thesis, which never outputs a message if the same message is already travelling. The following definition formalises this property.

Definition 3.2.3

Let $A_1, A_2, \ldots, A_n$ be serial FSMs with pairwise matching action signatures. Let $A_\|$ be the asynchronous parallel composition of all these FSMs.

The composition $A_\|$ is said to be *1-safe*, iff for all reachable states $q \in Q^{A_\|}$ it holds that $\forall x \in \pi_{n+1}(q).\ \pi_{n+1}(q)(x) = 1$.

When proving properties of composed automata, it is advantageous to consider only the interleaving of the component automata and derive results about parallel behaviour therefrom. However, this is only possible if the parallel composition behaves in a *confluent* way, that is, different scheduling of the components does not lead to different system states. Indeed the composition defined in Definition 3.2.2 is confluent. A weaker claim only considering serial FSMs suffices for all proofs later on, however.

Lemma 3.2.1

Let $A_1, A_2, \ldots, A_n$ be serial FSMs with pairwise matching action signatures. Let $A_\|$ be their asynchronous parallel composition.

Let $I \subseteq \Sigma_I^{A_\|} \cup \Sigma_\tau^{A_\|}$ and $O \subseteq \Sigma_O^{A_\|}$.

If $q \xrightarrow{I;O}_{A_\|} q''$ then either $|I| = 1$ or for all $i \in I$ there exists some $O' \subseteq O$ and a q' such that $q \xrightarrow{\{i\};O'}_{A_\|} q' \xrightarrow{I\setminus\{i\};O\setminus O'}_{A_\|} q''$.

Proof (Sketch)
See Isabelle/HOL for a formal version.

The action i must have originated from some component A_i. Taking O' to be O_i from Definition 3.2.2, the two steps are possible. $\square$

The parallel composition of FSMs is associative and commutative up to isomorphism.

Proposition 3.2.1
Let A, A' and A'' be state machines with pairwise matching action signatures.

$$\begin{aligned} A\|A' &\approx A'\|A \\ A\|(A'\|A'') &\approx A\|A'\|A'' \\ (A\|A')\|A'' &\approx A\|A'\|A'' \end{aligned}$$

Proof (Sketch)
See Isabelle/HOL for a formal version of commutativity.

Commutativity via

$$\varphi(q_1, q_2, M) = (q_2, q_1, M)\ .$$

Associativity via

$$\varphi(q_1, (q_2, q_3, M_1), M_2) = (q_1, q_2, q_3, M_1 + M_2)$$

and

$$\varphi((q_1, q_2, M_1), q_3, M_2) = (q_1, q_2, q_3, M_1 + M_2)$$

respectively. $\square$

4 Distributed Systems and Branching Time

4.1 Why It Should Not Work

The intuition why distributed implementations of arbitrary systems are impossible under branching-time semantics is easy to convey using a simple example. Consider the situation in Figure 4.1. A team of two robots stands in front of two doors. The robots want to reach at least one barrel of oil, but are separated from the barrels by two doors, which open and close. Clearly, if the two doors stay closed forever, the robots stand no chance, hence the restriction is imposed that at each point in time at least one door is open. As branching-time semantics are considered, it is assumed that the doors may close instantaneously at any point in time. Nonetheless there is a simple and robust protocol for the two robots to follow: Drive forward until the barrel is reached. As one door will be open at every point in time, one robot is guaranteed to drive through. Even if only one door ever opens, the two robot team reaches one barrel.

Compare now the situation in Figure 4.2 where the same two robots have been reused, but their batteries have been depleted from earlier usage and they cannot move until they have reloaded their batteries from an external source. Just such a source has been provided in form of an external battery right in the middle of the robots, containing enough charge to carry either robot to the respective barrel, but not both of them. Thus this example contains a distributed system consisting of two robots which need to communicate about which one gets to load its battery and moves. Once this has been decided, the branching-time assumption strikes: Whenever the charge has been transferred to some robot, say the upper one, the door in front of it closes. As the doors may move arbitrarily fast this can happen before the robot has any chance to move. Hence any forward movement by the upper robot is inhibited. Even if the two robots suspect that the upper door will not open and transfer the charge to the lower robot, the doors may switch status again and the lower door stays closed from then on. Continuing in this manner, no progress is ever made.

These considerations do not however exclude a randomised solution. As long as the behaviour of the doors is not all-knowing and downright evil, the robots stand a fair chance: By transferring the charge randomly between the two robots and trying to move every so often (note that in this idealised example world, no energy is lost if a move was unsuccessful), one robot will eventually manage to get past the respective door. As the time until this strategy succeeds is unknown a priori, branching-time equivalences often do detect a difference between this behaviour and the instantly successful attempt of Figure 4.1. If the equivalence in question does not, a randomised strategy, including

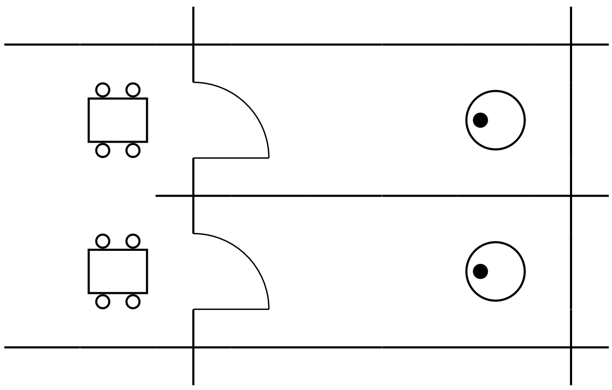

Figure 4.1: Two robots wanting to reach a barrel

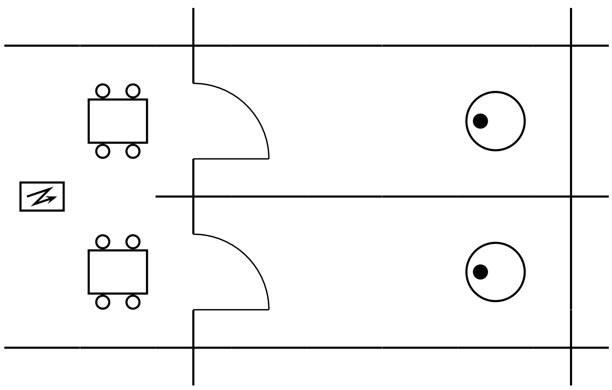

Figure 4.2: The same situation as in Figure 4.1 but with depleted batteries

an infinitely improbable infinite loop, is perfectly fine.

The equivalence notion used in the remainder of this section does not allow such loops. It identifies two systems if after the same observable behaviour, the two systems offer the same multisets of actions for execution. As the systems cannot enforce the execution of actions, but have to hope for the world to allow them, "offer" is probably the best choice of words here.

Definition 4.1.1
Let $N = (S^N, T^N, F^N, M_0^N, \ell^N)$ be a net, $\sigma \in \text{Act}^*$ and $X \subseteq \mathcal{M}(\text{Act})$.

$\langle \sigma, X \rangle$ is a *step ready pair* of N iff

$$\exists M.\ M_0^N \stackrel{\sigma}{\Longrightarrow}_N M \wedge M \stackrel{\tau}{\not\longrightarrow}_N \wedge X = \{A \in \mathcal{M}(\text{Act}) \mid M \stackrel{A}{\longrightarrow}_N\}\,.$$

The set of all step ready pairs of N is denoted $\mathscr{R}(N)$. Two nets N and N' are said to be *step readiness equivalent*, $N \approx_{\mathscr{R}} N'$, iff $\mathscr{R}(N) = \mathscr{R}(N')$.

4.2 Why It Does Not Work

Taking the formal definition of "distributed" from Section 2, it has already been proven that some behaviours cannot be implemented in a distributed way in [7]. This section will give a short recounting of the reasoning used there.

Unfortunately the intuitive example given at the beginning of this section does not map to the formal problem. The two robot system of Figure 4.2 can be represented as depicted in Figure 4.3 using a net. Using the formal definition of distributed, one finds that the system is already distributed, as the two transitions cannot fire in parallel. As no parallelism between transitions is needed, co-locating the two transitions would be a valid implementation. This would amount to connecting both robots to the external battery at once, placing them directly in front of the doors, and then trying to move forward. In that implementation, once a robot detects that it got past the door, it gets all the battery charge and moves to the goal. Assuming that the short moment while a robots futilely drives against a closed door consumes only a negligible amount of energy, this solves the problem.

However, such an implementation is not feasible in the situation depicted in Figure 4.4. The three robots try to reach at least two barrels, again having to reload their batteries from the two external batteries provided. For the sake of example the middle robot is twice as big as the other two, hence in need of twice the energy as well. As before, the doors open and close arbitrarily fast and unpredictably. The robots have two options to reach their goal of fetching two barrels. Either the upper and lower robot each grab one battery, move through the respective doors in front of them and reach one barrel each, or the larger robot in the middle grabs both batteries, moves through its door and reaches the two barrels.

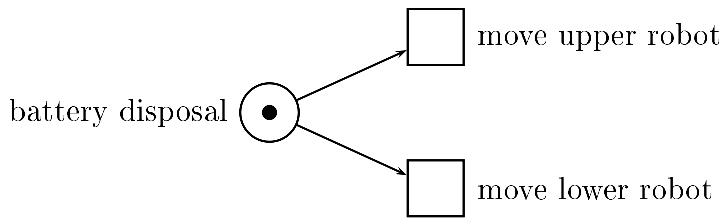

Figure 4.3: An abstract model of the situation in Figure 4.2

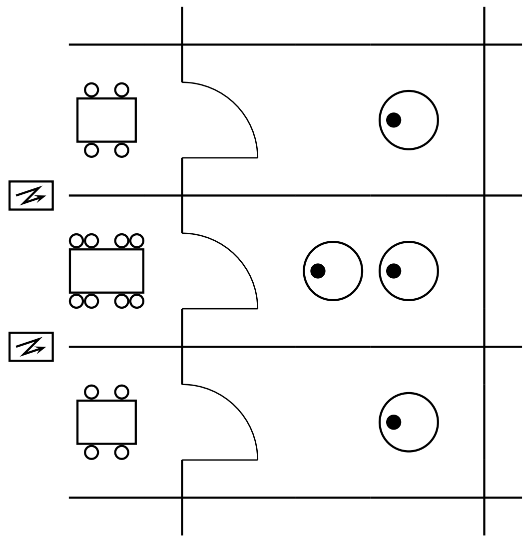

Figure 4.4: Three exhausted robots work in a team to reach a total of two barrels

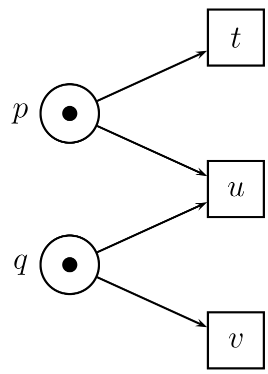

Figure 4.5: A fully reachable visible pure M

This robot problem corresponds to the net in Figure 4.5. As t and v can potentially happen in parallel they must not be co-located, hence at least one battery cannot be connected to both neighbouring robots, giving rise to the same problems as before. In [7] we found the structure depicted in these figures to be at the core of the problem. The structure can be described formally as follows.

Definition 4.2.1
Let $N = (S^N, T^N, F^N, M_0^N, \ell^N)$ be a net. N has a *fully reachable visible pure* M iff

$$\begin{aligned}\exists t, u, v \in T^N.\ & {}^\bullet t \cap {}^\bullet u \neq \emptyset \wedge {}^\bullet u \cap {}^\bullet v \neq \emptyset \wedge {}^\bullet t \cap {}^\bullet v = \emptyset \wedge \\ & \ell^N(t) \neq \emptyset \wedge \ell^N(u) \neq \emptyset \wedge \ell^N(v) \neq \emptyset \wedge \\ & \exists M \in [M_0^N\rangle.\ {}^\bullet t \cup {}^\bullet u \cup {}^\bullet v \subseteq M\ .\end{aligned}$$

Clearly, a net containing a fully reachable visible pure M cannot be distributed. Trying to implement such a net in a distributed manner, one quickly finds that a fully reachable visible pure M gives rise to a particular step ready pair.

Proposition 4.2.1
Let $N = (S^N, T^N, F^N, M_0^N, \ell^N)$ be a plain net which has a fully reachable visible pure M. There exists $\langle \sigma, X \rangle \in \mathscr{R}(N)$ with

$$\exists a, b, c \in \text{Act}.\ a \neq c \wedge \{b\} \in X \wedge \{a, c\} \in X \wedge \{a, b\} \notin X \wedge \{b, c\} \notin X\ .$$

Proof
See [7]. □

In order to implement a net exhibiting such a step ready pair, one needs at least three transitions executing the three different actions a, b, and c. As the set X describes the possible sets of actions after a certain marking M has been reached, all three transitions must be enabled in the same marking M. Furthermore the transitions executing a and c can happen in parallel and hence cannot share a preplace due to Definition 3.1.1. The transitions executing a and b cannot execute together, so some shared preplace must exist. The same holds for the pair of b and c. The transition and place structure just described sounds familiar. Indeed the transitions executing a, b, and c are guaranteed to form a fully reachable visible pure M.

Proposition 4.2.2
Let $N = (S^N, T^N, F^N, M_0^N, \ell^N)$ be a net such that there exists $\langle \sigma, X \rangle \in \mathscr{R}(N)$ with $\exists a, b, c \in \text{Act}.\ a \neq c \wedge \{b\} \in X \wedge \{a, c\} \in X \wedge \{a, b\} \notin X \wedge \{b, c\} \notin X$. Then N has a fully reachable visible pure M.

Proof
See [7]. □

From these propositions, it follows that no distributed system can exhibiting the same behaviour as the system of Figure 4.5 up to step readiness equivalence. Hence not all system behaviours can be implemented in a distributed fashion if step readiness equivalence is used to compare systems. This result depends on two properties of step readiness equivalence which are not necessary for branching-time equivalences in general. Step readiness equivalence does not allow the implementation to use divergence, hence a randomised implementation is ruled out. Furthermore step readiness equivalence respects parallelism. Otherwise the system could be stripped of all its parallelism by introducing a new place connected to all transitions by a loop. After all parallelism has been removed the trivial distribution, co-locating all elements is allowed by Definition 3.1.1. Apart from that however, step readiness equivalence is quite a coarse branching-time equivalence, hence the impossibility of implementing fully reachable visible pure Ms should hold for most branching-time equivalences.

5 Distributed Systems and Linear Time

5.1 Why It Should Work

As daily progress in design and deployment of distributed systems shows, there must be some way for distributed systems to do useful work in the real world. So either there exists no real world demand for the behaviours identified as problematic in the previous section, or the branching-time assumption is not always warranted.

It is indeed the second possibility as a short example demonstrates. Consider a web shop which sells small four wheeled robots to computer scientists. At some point a scientist has decided to buy a robot. Then the web shop software and some software of the scientist's bank will communicate to ensure prompt payment. The system consisting of these two software agents has basically two options. Either both agree that the money shall be transferred and the robot shall be sent. Or they agree on not performing the transaction, usually due to insufficiency of either robots or, more likely, money. They comprise a distributed system and can only communicate asynchronously. However no branching-time problems can arise. The scientist, after having triggered the "buy" button, is simply not offered any means to communicate a possible change of mind to the web shop software, and the bank software will blissfully ignore possible concurrent withdrawals and produce overdraft. Thus while the communication between web shop and bank is in progress, the environment cannot change in ways which will make the execution of either option impossible.

Hence this sections considers linear-time semantics. The system is assumed to be fast in comparison to the world and can first measure all relevant aspect of the world and therefrom infer which actions will be possible later. Returning to the example from the earlier section, consider again the robots in Figure 4.2. If the doors are slow in comparison to the robots' thoughts, the solution is fairly straightforward. Each robots monitors the status of the door in front of it. Once a door opens, the charge is transferred to the robot standing in front of it. The robot subsequently moves before the door has closed again, thus solving the problem. Ignoring the explicit door monitoring step, this can be modelled abstractly by assuming that every action the system makes is indeed possible, as otherwise the system would not have chosen to execute that action in the first place.

Note that this "correctness" of choices is not explicitly represented in the formal models under consideration. Rather, the difference is in the equivalence relation used for comparing two systems. Earlier two systems were only equivalent if at each indistinguishable point of execution they offered the same set of actions to the world, i.e. would react the same to any states of the world. Now however, two systems are already equivalent if both offer the same set of possible execution sequences. As both systems are assumed intelligent enough to make the right choices every time, they would make the same choices in the same situation and hence exhibit the same behaviour as well.

Also, the equivalence relation will discern live- and deadlocks of the implementation, in particular since distributed systems have a proven tendency to exhibit them. To prove that the construction given later does not introduce new live- or deadlocks, an equivalence which notices those is necessary. Finally, the equivalence notion will discern differences in parallelism, i.e. two systems of which only one can do two particular actions in parallel are different. This requirement helps discern systems of different performance.

Definition 5.1.1

Let $N = (S^N, T^N, F^N, M_0^N, \ell^N)$ be a net, $\sigma \in \mathcal{M}(\mathrm{Act})^*$ and $0, \delta \notin \mathrm{Act}$.

σ is an *incomplete step trace* of N iff

$$\exists M \subseteq S^N.\ M_0^N \overset{\sigma}{\Longrightarrow}_N M\ .$$

$\sigma 0$ is a *completed step trace of* of N iff

$$\exists M \subseteq S^N.\ M_0^N \overset{\sigma}{\Longrightarrow}_N M \land M \overset{\tau}{\not\longrightarrow}_N \land \nexists A.\ M \overset{A}{\longrightarrow}_N\ .$$

$\sigma\delta$ is a *diverging step trace* of N iff

$$\exists M \subseteq S^N.\ M_0^N \overset{\sigma}{\Longrightarrow}_N M \land M \overset{\tau}{\longrightarrow}_N \overset{\tau}{\longrightarrow}_N \overset{\tau}{\longrightarrow}_N \cdots\ .$$

The set of all incomplete, completed, and diverging step traces of N is denoted $\mathrm{CST}(N)$. Two nets N and N' are said to be *completed step trace equivalent*, iff $\mathrm{CST}(N) = \mathrm{CST}(N')$.

Completed step trace equivalence is a straightforward extension of the well known completed trace equivalence. In particular, it adds the ability to detect parallelism but does not discern different causal structures. Like completed trace equivalence it does not detect deadlocks in one component of a system, as long as some activity can continue. Also similarly, it does not imply any fairness or justness conditions. It detects livelocks even if they are completely independent of other activities in the system, however. Also, this equivalence mirrors my intuition that if a system can perform activities in parallel, it does not need to perform them in parallel every time, but will do so often enough to make the performance improvement significant.

After having defined two systems to be equivalent as per Definition 5.1.1, the remaining task is to give an algorithm which, given an arbitrary net, constructs an equivalent distributed version of it. The main problem it solves is how to make a coherent choice of actions in a set of partly parallel, partly conflicting transitions. In contrast to the results in Section 4, this choice can be made arbitrary early, in particular without actually firing any of the transitions. Why is this so? Because it is assumed that all relevant information about the world is already known to make the correct choice. Hence the transitions in question will first reach a consensus about which ones fire without exhibiting any external behaviour and then execute the preplanned set of transitions later. Details of how that works are given below.

5.2 How It Does Work

This section contains the main results of this thesis and gives a constructive proof of the existence of a distributed implementation for every behaviour representable by a plain net up to completed step trace equivalence.

The proof will start at an arbitrary plain net, transforming it into a network of communicating serial FSMs. Each serial FSM will in turn be transformed into a net, and similarly the coupling between the FSMs will also be transformed into net structures. This slightly indirect approach allows the interesting problems of the distribution protocol to be described in the more compact model of the FSMs. The second mapping, from FSMs to nets, will be very direct, thereby carrying over the correctness of the protocol back into the domain of Petri nets.

Before delving into the formal definitions, the intuition behind the protocol should be explained. Assume a net N is given. First an arbitrary but fixed total order over all places of N is defined. Then places and transitions of N will be replaced, or *implemented*, by small subnets which only communicate asynchronously.

The implementation of a transition, say t, waits until all preplaces of t have received a token. When it decides to fire, the implementation of t requests exclusive permission to use a token from (*locks*) all its preplaces in that global order. While the lock is not acquired, no further activity occurs in the implementation of t. The global order guarantees that deadlocks do not occur. Assume the greatest (according to the global order) locked place is p, then the transition holding the lock on place p will only attempt to acquire locks on places greater than p. Once the implementation of t holds locks on all preplaces of t, it fires, notifies the preplaces of the token removal, and produces new tokens on all postplaces.

The main complication is handling of failed lock attempts. When the implementation of a transition t was waiting to acquire a lock on a place p, yet another transition u succeeded in firing and removed the token located on p, the implementation of t must abort the lock attempt, must release all currently held locks and resume waiting for all preplaces to become marked. Livelocks do not occur, as whenever transition t fails to acquire a lock, some other transition must have fired.

The rest of the algorithm is basically bookkeeping.

The protocol between places and transitions uses the following messages, which all carry indices denoting the communication partners:

- notify_s^t (*place s has received a token*)
- $\mathrm{success}_s^t$ (*place s granted the lock to transition t*)
- loose_s^t (*some transition different from t locked the place s and removed the token from it*)
- token_s^t (*place s acknowledges the removal of its token by the transition t*)
- lock_s^t (*transition t requests exclusive permission to use the token on place s*)
- ackU_s^t (*transition t acknowledges the removal of the token on place s, while no locking request is pending from t to s*)
- ackL_s^t (*transition t acknowledges the removal of the token on place s, after a locking request has been sent to s*)
- unlock_s^t (*transition t releases the lock on place s*)
- go_s^t (*transition t removes the token from s*)
- $\mathrm{newToken}_s^t$ (*transition t produces a new token on s*)

First, the implementation of transitions will be given as an FSM. The implementation operates in two phases. The first phase collects information about which preplaces are marked and starts to lock preplaces once all are marked. The second phase is the actual firing, notifying all preplaces about the removal of a token, then waiting until all preplaces have acknowledged said removal. Finally new tokens are produced on the postplaces.

The internal actions used are as follows:

- $\mathrm{internalLock}_l^t$ (*transition t starts to lock place l*)
- $\mathrm{internalFire}^t$ (*transition t begins firing and starts to remove tokens from preplaces*)
- $\mathrm{internalDone}_l^t$ (*transition t has finished firing and produces tokens on postplaces*)

The states of the implementation mirror the two phases closely:

- $\mathrm{locking}_t(L, l, T)$ (*The transition t tries to lock preplaces. All preplaces in T currently hold a token, preplaces in L have already been locked, the lock on preplace l is currently being acquired. If $l = \bot$ no lock is currently being acquired.*)
- $\mathrm{firing}_t(T)$ (*The transition t removes tokens from the preplaces. Tokens from the preplaces in T have already arrived.*)

Definition 5.2.1

Let $N = (S^N, T^N, F^N, M_0^N, \ell^N)$ be a plain net. Let $\leq$ be a total order over S^N. Let $\bot \notin T$ be some new object.

For every transition $t \in T^N$ the *transition simulating automaton* of t is defined as an FSM $A_t = (\Sigma^{A_t}, Q^{A_t}, q_0^{A_t}, \rightarrow^{A_t})$ with

- $\Sigma^{A_t} = (\Sigma_I^{A_t}, \Sigma_O^{A_t}, \Sigma_\tau^{A_t})$ with
  - $\Sigma_I^{A_t} = \{\text{notify}_s^t, \text{success}_s^t, \text{loose}_s^t, \text{token}_s^t \mid s \in {}^\bullet t\}$,
  - $\Sigma_O^{A_t} = \{\text{lock}_s^t, \text{ackU}_s^t, \text{ackL}_s^t, \text{unlock}_s^t, \text{go}_s^t \mid s \in {}^\bullet t\} \cup$ $\{\text{newToken}_s^t \mid s \in t^\bullet\} \cup$ $\{\text{fire}^t\}$,
  - $\Sigma_\tau^{A_t} = \{\text{internalLock}_l^t, \text{internalDone}_l^t, \text{internalFire}^t\}$,
- $Q^{A_t} = \{\text{locking}_t(L, l, T) \mid L, T \subseteq {}^\bullet t, l = \bot \vee l \in {}^\bullet t\} \cup \{\text{firing}_t(T) \mid T \subseteq {}^\bullet t\}$,
- $q_0^{A_t} = \text{locking}_t(\emptyset, \bot, \emptyset)$,

and $\rightarrow^{A_t}$ such that

- $\text{locking}_t(L, l, T) \xrightarrow{\{\text{notify}_s^t\};\emptyset}_{A_t} \text{locking}_t(L, l, T \cup \{s\})$ for each $s \notin T$,
- $\text{locking}_t(L, l, T) \xrightarrow{\{\text{loose}_s^t\};\{\text{ackU}_s^t\}}_{A_t} \text{locking}_t(L, l, T \setminus \{s\})$ for $s \in T \setminus L, s \neq l \neq \bot$,
- $\text{locking}_t(L, \bot, T) \xrightarrow{\{\text{loose}_s^t\};\{\text{ackU}_s^t\} \cup \left\{\text{unlock}_p^t \mid p \in L\right\}}_{A_t} \text{locking}_t(\emptyset, \bot, T \setminus \{s\})$ for $s \in T \setminus L$,
- $\text{locking}_t(L, l, T) \xrightarrow{\{\text{loose}_l^t\};\{\text{ackL}_l^t\} \cup \left\{\text{unlock}_p^t \mid p \in L\right\}}_{A_t} \text{locking}_t(\emptyset, \bot, T \setminus \{l\})$,
- $\text{locking}_t(L, \bot, {}^\bullet t) \xrightarrow{\{\text{internalLock}_l^t\};\{\text{lock}_l^t\}}_{A_t} \text{locking}_t(L, l, {}^\bullet t)$ for $l = \min({}^\bullet t \setminus L)$,
- $\text{locking}_t(L, l, {}^\bullet t) \xrightarrow{\{\text{success}_l^t\};\emptyset}_{A_t} \text{locking}_t(L \cup \{l\}, \bot, {}^\bullet t)$,
- $\text{locking}_t(L, l, T) \xrightarrow{\{\text{success}_l^t\};\left\{\text{unlock}_p^t \mid p \in L \cup \{l\}\right\}}_{A_t} \text{locking}_t(\emptyset, \bot, T)$ for each $T \neq {}^\bullet t$,
- $\text{locking}_t({}^\bullet t, \bot, {}^\bullet t) \xrightarrow{\{\text{internalFire}^t\};\{\text{fire}^t\} \cup \left\{\text{go}_s^t \mid s \in {}^\bullet t\right\}}_{A_t} \text{firing}_t(\emptyset)$,
- $\text{firing}_t(T) \xrightarrow{\{\text{token}_s^t\};\emptyset}_{A_t} \text{firing}_t(T \cup \{s\})$ for each $s \notin T$, and
- $\text{firing}_t({}^\bullet t) \xrightarrow{\{\text{internalDone}^t\};\left\{\text{newToken}_s^t \mid s \in t^\bullet\right\}}_{A_t} \text{locking}_t(\emptyset, \bot, \emptyset)$.

The implementation of a place goes through the following phases: First the place is empty, and the implementation is not sending anything. Then a token arrives and the implementation notifies all posttransitions. Then the place gets locked by some posttransition, possibly queueing other locking requests until the lock holding transition succeeds in firing or releases the lock. If the lock is released another transition from the queue is immediately granted the lock. If the current lock holder succeeds in firing, all other transitions are notified of the token removal. Then the implementation enters its fourth phase waiting for all transitions to acknowledge said removal, possibly clearing pending lock requests on the way.

The internal actions used are as follows:

- internalNotify^s (*place s notifies its posttransitions about the arrival of a token*)
- $\text{internalPassToken}_s^t$ (*place s sends its token to the transition t*)

The states of the implementation mirror the phases as follows:

- empty_s (*Place s is empty.*)
- prenotify_s (*Place s holds a token but has not yet notified its posttransitions.*)
- unlocked_s (*Place s holds a token, has notified its posttransitions but is not yet locked.*)
- $\text{locked}_s(t, L)$ (*Place s is locked by transition t, the transitions in L also sent a lock request but have not been granted the lock.*)

- $\text{waiting}_s(t, L, W)$ (*The token on place s needs to travel to the transition t, lock requests from all transitions in L have been received, token removal acknowledgements from all transitions in W have not yet arrived.*)

Definition 5.2.2

Let $N = (S^N, T^N, F^N, M_0^N, \ell^N)$ be a plain net.

For every place $s \in S^N$ the *place simulating automaton* of s is defined as an FSM $A_s = (\Sigma^{A_s}, Q^{A_s}, q_0^{A_s}, \rightarrow^{A_s})$ with

- $\Sigma^{A_s} = (\Sigma_I^{A_s}, \Sigma_O^{A_s}, \Sigma_\tau^{A_s})$ with
  - $\Sigma_I^{A_s} = \{\text{lock}_s^t, \text{ackU}_s^t, \text{ackL}_s^t, \text{unlock}_s^t, \text{go}_s^t \mid t \in s^\bullet\} \cup \{\text{newToken}_s^t \mid t \in {}^\bullet s\}$,
  - $\Sigma_O^{A_s} = \{\text{notify}_s^t, \text{success}_s^t, \text{loose}_s^t, \text{token}_s^t \mid t \in s^\bullet\}$,
  - $\Sigma_\tau^{A_s} = \{\text{internalNotify}^s\} \cup \{\text{internalPassToken}_s^t \mid t \in s^\bullet\}$,
- $Q^{A_s} = \{\text{empty}_s, \text{prenotify}_s, \text{unlocked}_s\} \cup \{\text{locked}_s(t, L) \mid t \in s^\bullet, L \subseteq s^\bullet, t \notin L\} \cup \{\text{waiting}_s(t, L, W) \mid t \in s^\bullet, W \subseteq s^\bullet, t \notin W, L \subseteq W\}$,
- $q_0^{A_s} = \begin{cases} \text{prenotify}_s & \text{if } s \in M_0^N \\ \text{empty}_s & \text{otherwise} \end{cases}$,

and $\rightarrow^{A_s}$ such that

- $\text{empty}_s \xrightarrow{\{\text{newToken}_s^t\};\emptyset}_{A_s} \text{prenotify}_s$,
- $\text{prenotify}_s \xrightarrow{\{\text{internalNotify}^s\};\{\text{notify}_s^t \mid t \in s^\bullet\}}_{A_s} \text{unlocked}_s$,
- $\text{unlocked}_s \xrightarrow{\{\text{lock}_s^t\};\{\text{success}_s^t\}}_{A_s} \text{locked}_s(t, \emptyset)$,
- $\text{locked}_s(t, L) \xrightarrow{\{\text{lock}_s^u\};\emptyset}_{A_s} \text{locked}_s(t, L \cup \{u\})$ for each $u \neq t, u \notin L$,
- $\text{locked}_s(t, L) \xrightarrow{\{\text{unlock}_s^t\};\{\text{success}_s^u\}}_{A_s} \text{locked}_s(u, L \setminus \{u\})$ for each $u \in L$,
- $\text{locked}_s(t, \emptyset) \xrightarrow{\{\text{unlock}_s^t\};\emptyset}_{A_s} \text{unlocked}_s$,
- $\text{locked}_s(t, L) \xrightarrow{\{\text{go}_s^t\};\{\text{loose}_s^u \mid u \in s^\bullet, u \neq t\}}_{A_s} \text{waiting}_s(t, L, s^\bullet \setminus \{t\})$,
- $\text{waiting}_s(t, L, W) \xrightarrow{\{\text{lock}_s^u\};\emptyset}_{A_s} \text{waiting}_s(t, L \cup \{u\}, W)$ for each $u \notin L, u \in W$,
- $\text{waiting}_s(t, L, W) \xrightarrow{\{\text{ackL}_s^u\};\emptyset}_{A_s} \text{waiting}_s(t, L \setminus \{u\}, W \setminus \{u\})$ for each $u \in L$,
- $\text{waiting}_s(t, L, W) \xrightarrow{\{\text{ackU}_s^u\};\emptyset}_{A_s} \text{waiting}_s(t, L, W \setminus \{u\})$ for each $u \notin L, u \in W$, and
- $\text{waiting}_s(t, \emptyset, \emptyset) \xrightarrow{\{\text{internalPassToken}_s^t\};\{\text{token}_s^t\}}_{A_s} \text{empty}_s$.

Definition 5.2.3

Let $N = (S^N, T^N, F^N, M_0^N, \ell^N)$ be a plain net.

The *FSM based asynchronous implementation* of N, A_N, is given by

$$A_N = \Big\|_{x \in S^N \cup T^N} A_x \,.$$

A proof that the construction from Definition 5.2.1, Definition 5.2.2, and Definition 5.2.3 is correct, would need a clear notion of correctness. Instead of redefining completed step trace equivalence for state machines however, the following gives behavioural properties of the implementation which will ultimately be used in Theorem 5.2.1 to show completed step trace equivalence for the overall transformation.

The first interesting property concerns the reachable state space of implementations of transitions.

Lemma 5.2.1

Let $N = (S^N, T^N, F^N, M_0^N, \ell^N)$ be a plain net, let $\leq$ be a total order over S^N, and let $t \in T^N$. Let A_t be the transition simulating automaton of t.

Let q be a reachable state of A_t.

Then $\beta(q)$ with

$$\beta(q) \Leftrightarrow q \in \left\{ \text{locking}_t(L, l, T) \;\middle|\; \begin{array}{l} L \subseteq T \subseteq {}^\bullet t, \forall s \in L, p \in {}^\bullet t \setminus L.\ s < p, \\ L = \emptyset \vee l \neq \bot \vee T = {}^\bullet t, \\ l = \bot \vee \\ (l \in T \wedge \forall s \in L, p \in {}^\bullet t \setminus (L \cup \{l\}).\ s < l < p) \end{array} \right\} \cup \{\text{firing}_t(T) \mid T \subseteq {}^\bullet t\}$$

Proof

Via induction over the steps necessary to reach q.

$\beta(q_0^{A_t})$ is trivial.

Let q, I, O, and q' such that $q \xrightarrow{I;O}_{A_t} q'$ with $\beta(q)$. The proof of $\beta(q')$ happens via case distinction over the performed step.

Case $\text{locking}_t(L, l, T) \xrightarrow{\{\text{notify}_s^t\};\emptyset}_{A_t} \text{locking}_t(L, l, T \cup \{s\})$, $s \notin T$: Only T changed, and it became larger.

Case $\text{locking}_t(L, l, T) \xrightarrow{\{\text{loose}_s^t\};\{\text{ackU}_s^t\}}_{A_t} \text{locking}_t(L, l, T \setminus \{s\})$, $s \in T, s \notin L, s \neq l \neq \bot$: As only s was removed from T and $s \notin L$ still $L \subseteq T \subseteq {}^\bullet t$. Also $l \neq \bot$ hence still $L \neq \emptyset \vee l \neq \bot \vee T = {}^\bullet t$. And $s \neq l$ thus still $l \in T$.

Case $\text{locking}_t(L, \bot, T) \xrightarrow{\{\text{loose}_s^t\};\{\text{ackU}_s^t\} \cup \{\text{unlock}_p^t \mid p \in L\}}_{A_t} \text{locking}_t(\emptyset, \bot, T \setminus \{s\})$, $s \in T \setminus L$: All conditions are trivial.

Case $\text{locking}_t(L, l, T) \xrightarrow{\{\text{loose}_l^t\};\{\text{ackL}_l^t\} \cup \{\text{unlock}_p^t \mid p \in L\}}_{A_t} \text{locking}_t(\emptyset, \bot, T \setminus \{l\})$: All conditions are trivial.

Case $\text{locking}_t(L, \bot, {}^\bullet t) \xrightarrow{\{\text{internalLock}_l^t\};\{\text{lock}_l^t\}}_{A_t} \text{locking}_t(L, l, {}^\bullet t)$, $l = \min({}^\bullet t \setminus L)$: As the l was chosen to be the minimum of ${}^\bullet t \setminus L$ clearly $l \in {}^\bullet t$ and with the additional fact that $\forall s \in L, p \in {}^\bullet t \setminus L.\ s < p$ also $\forall s \in L, p \in {}^\bullet t \setminus (L \cup \{l\}).\ s < l < p$.

Case $\text{locking}_t(L, l, {}^\bullet t) \xrightarrow{\{\text{success}_l^t\};\emptyset}_{A_t} \text{locking}_t(L \cup \{l\}, \bot, {}^\bullet t)$: From $l \in {}^\bullet t$ follows that after the step $L \cup \{l\} \subseteq {}^\bullet t$ and from $\forall s \in L, p \in {}^\bullet t \setminus (L \cup \{l\}).\ s < l < p$ follows that $\forall s \in L \cup \{l\}, p \in {}^\bullet t \setminus (L \cup \{l\}).\ s < p$. The rest is trivial.

Case $\text{locking}_t(L, l, T) \xrightarrow{\{\text{success}_l^t\};\{\text{unlock}_p^t \mid p \in L \cup \{l\}\}}_{A_t} \text{locking}_t(\emptyset, \bot, T)$, $T \neq {}^\bullet t$: All conditions are trivial.

Case $\mathrm{locking}_t({}^\bullet t, \perp, {}^\bullet t) \xrightarrow{\{\mathrm{internalFire}^t\};\{\mathrm{fire}^t\}\cup\left\{\mathrm{go}_s^t \;\middle|\; s\in{}^\bullet t\right\}}_{A_t} \mathrm{firing}_t(\emptyset)$: Trivial.

Case $\mathrm{firing}_t(T) \xrightarrow{\{\mathrm{token}_s^t\};\emptyset}_{A_t} \mathrm{firing}_t(T \cup \{s\})$, $s \notin T$: Trivial.

Case $\mathrm{firing}_t({}^\bullet t) \xrightarrow{\{\mathrm{internalDone}^t\};\left\{\mathrm{newToken}_s^t \;\middle|\; s\in t^\bullet\right\}}_{A_t} \mathrm{locking}_t(\emptyset, \perp, \emptyset)$: All conditions again trivial. □

To shorten the following formulae somewhat, the tuples constituting the composed state machine states will be equipped with a $\tilde{\in}$ operator as follows. If q is a tuple of length $n+1$, $x \mathrel{\tilde{\in}} q$ iff $\exists i \leq n.\ \pi_i(q) = x \vee x \in \pi_{n+1}(q)$. Per construction x will always carry some indices denoting an original transition or place which uniquely determine the only index in q where it could possibly occur. Also, keep in mind that the last element of the state-tuple of the composed FSMs is the message buffer. Thus $x \mathrel{\tilde{\in}} q$ basically means "the component denoted by the indices of x is in the state x" or "the message x is currently travelling" depending on whether x is a message or a state.

Another property of the transformation consists of two mappings between the states of the composed state machine and those of the original net. In both mappings the states $\mathrm{prenotify}_s$, $\mathrm{unlocked}_s$ and locked_s correspond to full places, whereas all other states correspond to empty places, except for the duration of transition firings. While in the original net a transition fires with instantaneous effects, the firing of a transition is a lengthy process in the implementation. The first mapping $\mathfrak{f}$ is coherent with the observable actions, i.e. changes the marking mapped to at the same time as an observable action is performed and maps to a marking where all currently firing transitions have completely fired. The second mapping $\mathfrak{f}'$ maps similarly but only considers transitions which left their $\mathrm{firing}_t(T)$ phase completed. While this mapping is not coherent with the observed actions, it helps with the proof of correctness. In particular it carries the contact freeness of the original net into the implementation in such a way that the contact freeness becomes available as an argument at the point where a transition finishes firing.

Definition 5.2.4

Let N be a plain net and let A_N be the FSM based implementation of it.

The function $\mathfrak{f}:\ Q^{A_N} \to \mathcal{P}(S^N)$ is defined as

$$\mathfrak{f}(q) = \left\{ s \in S^N \;\middle|\; \begin{array}{l} (\nexists t.\ \mathrm{go}_s^t \mathrel{\tilde{\in}} q \wedge (\mathrm{prenotify}_s \mathrel{\tilde{\in}} q \vee \mathrm{unlocked}_s \mathrel{\tilde{\in}} q \vee \\ \exists t, L.\ \mathrm{locked}_s(t, L) \mathrel{\tilde{\in}} q \vee \exists t.\ \mathrm{newToken}_s^t \mathrel{\tilde{\in}} q)) \vee \\ \exists t \in {}^\bullet s, T.\ \mathrm{firing}_t(T) \mathrel{\tilde{\in}} q \end{array} \right\}.$$

The function $\mathfrak{f}':\ Q^{A_N} \to \mathcal{P}(S^N)$ is defined as

$$\mathfrak{f}'(q) = \left\{ s \in S^N \;\middle|\; \begin{array}{l} \mathrm{prenotify}_s \mathrel{\tilde{\in}} q \vee \mathrm{unlocked}_s \mathrel{\tilde{\in}} q \vee \\ \exists t, L.\ \mathrm{locked}_s(t, L) \mathrel{\tilde{\in}} q \vee \exists t.\ \mathrm{newToken}_s^t \mathrel{\tilde{\in}} q \vee \\ \exists t \in s^\bullet, T.\ \mathrm{firing}_t(T) \mathrel{\tilde{\in}} q \end{array} \right\}.$$

Some states of the state machine, although related through above functions with states of the net, are in fact never reached. A predicate is needed which decides whether an automaton state is actually a valid state. It will be proven later that only valid states are reachable in the automaton.

Definition 5.2.5

Let N be a plain net and let A_N be the FSM based implementation of it.

Let $n = |T^N| + |S^N|$.

The predicate $\alpha \subseteq Q^{A_N}$ is defined as $\alpha(q)$ iff

(A.a) $\mathfrak{f}(q) \in [M_0^N\rangle$,

(A.b) $\mathfrak{f}'(q) \in [M_0^N\rangle$,

(B) $\forall x.\ \pi_{n+1}(q)(x) \leq 1$,

(C.s) $\text{notify}_s^t \mathrel{\tilde{\in}} q \Rightarrow \text{unlocked}_s \mathrel{\tilde{\in}} q \lor$
$\exists u, L.\ \text{locked}_s(u, L) \mathrel{\tilde{\in}} q \land u \neq t \land t \notin L \lor$
$\exists u, L, W.\ \text{waiting}_s(u, L, W) \mathrel{\tilde{\in}} q \land u \neq t \land t \in W \land t \notin L$,

(C.t) $\text{notify}_s^t \mathrel{\tilde{\in}} q \Rightarrow \exists L, l, T.\ \text{locking}_t(L, l, T) \mathrel{\tilde{\in}} q \land s \notin T$,

(C.e) $\text{notify}_s^t \mathrel{\tilde{\in}} q \Rightarrow \text{success}_s^t \mathrel{\tilde{\notin}} q \land \text{token}_s^t \mathrel{\tilde{\notin}} q \land \text{lock}_s^t \mathrel{\tilde{\notin}} q \land \text{ackU}_s^t \mathrel{\tilde{\notin}} q \land$
$\text{ackL}_s^t \mathrel{\tilde{\notin}} q \land \text{unlock}_s^t \mathrel{\tilde{\notin}} q \land \text{go}_s^t \mathrel{\tilde{\notin}} q \land \nexists u.\ \text{newToken}_s^u \mathrel{\tilde{\in}} q$,

(D.s) $\text{success}_s^t \mathrel{\tilde{\in}} q \Rightarrow \exists L.\ \text{locked}_s(t, L) \mathrel{\tilde{\in}} q$,

(D.t) $\text{success}_s^t \mathrel{\tilde{\in}} q \Rightarrow \exists L, T.\ \text{locking}_t(L, s, T) \mathrel{\tilde{\in}} q$,

(D.e) $\text{success}_s^t \mathrel{\tilde{\in}} q \Rightarrow \text{notify}_s^t \mathrel{\tilde{\notin}} q \land \text{loose}_s^t \mathrel{\tilde{\notin}} q \land \text{token}_s^t \mathrel{\tilde{\notin}} q \land \text{lock}_s^t \mathrel{\tilde{\notin}} q \land \text{ackU}_s^t \mathrel{\tilde{\notin}} q \land$
$\text{ackL}_s^t \mathrel{\tilde{\notin}} q \land \text{unlock}_s^t \mathrel{\tilde{\notin}} q \land \text{go}_s^t \mathrel{\tilde{\notin}} q \land \nexists u.\ \text{newToken}_s^u \mathrel{\tilde{\in}} q$,

(E.s) $\text{loose}_s^t \mathrel{\tilde{\in}} q \Rightarrow \exists u, L, W.\ \text{waiting}_s(u, L, W) \mathrel{\tilde{\in}} q \land u \neq t \land t \in W$,

(E.t) $\text{loose}_s^t \mathrel{\tilde{\in}} q \Rightarrow \exists L, l, T.\ \text{locking}_t(L, l, T) \mathrel{\tilde{\in}} q \land s \in T \land s \notin L \lor$
$\text{notify}_s^t \mathrel{\tilde{\in}} q$,

(E.e) $\text{loose}_s^t \mathrel{\tilde{\in}} q \Rightarrow \text{success}_s^t \mathrel{\tilde{\notin}} q \land \text{token}_s^t \mathrel{\tilde{\notin}} q \land \text{ackU}_s^t \mathrel{\tilde{\notin}} q \land$
$\text{ackL}_s^t \mathrel{\tilde{\notin}} q \land \text{unlock}_s^t \mathrel{\tilde{\notin}} q \land \text{go}_s^t \mathrel{\tilde{\notin}} q \land \nexists u.\ \text{newToken}_s^u \mathrel{\tilde{\in}} q$,

(F.s) $\text{token}_s^t \mathrel{\tilde{\in}} q \Rightarrow \text{empty}_s \mathrel{\tilde{\in}} q$,

(F.t) $\text{token}_s^t \mathrel{\tilde{\in}} q \Rightarrow \exists T.\ \text{firing}_t(T) \mathrel{\tilde{\in}} q \land s \notin T$,

(F.e) $\text{token}_s^t \mathrel{\tilde{\in}} q \Rightarrow \text{notify}_s^t \mathrel{\tilde{\notin}} q \land \text{success}_s^t \mathrel{\tilde{\notin}} q \land \text{loose}_s^t \mathrel{\tilde{\notin}} q \land \text{lock}_s^t \mathrel{\tilde{\notin}} q \land \text{ackU}_s^t \mathrel{\tilde{\notin}} q \land$
$\text{ackL}_s^t \mathrel{\tilde{\notin}} q \land \text{unlock}_s^t \mathrel{\tilde{\notin}} q \land \text{go}_s^t \mathrel{\tilde{\notin}} q \land \nexists u.\ \text{newToken}_s^u \mathrel{\tilde{\in}} q$,

(G.s) $\text{lock}_s^t \mathrel{\tilde{\in}} q \Rightarrow \text{unlocked}_s \mathrel{\tilde{\in}} q \lor$
$\exists u, L.\ \text{locked}_s(u, L) \mathrel{\tilde{\in}} q \land u \neq t \land t \notin L \lor$
$\exists L.\ \text{locked}_s(t, L) \mathrel{\tilde{\in}} q \land \text{unlock}_s^t \mathrel{\tilde{\in}} q \lor$
$\exists u, L, W.\ \text{waiting}_s(u, L, W) \mathrel{\tilde{\in}} q \land u \neq t \land t \in W \land t \notin L$,

(G.t) $\text{lock}_s^t \mathrel{\tilde{\in}} q \Rightarrow \exists L, T.\ \text{locking}_t(L, s, T) \mathrel{\tilde{\in}} q \lor$
$\text{ackL}_s^t \mathrel{\tilde{\in}} q$,

$$
\begin{aligned}
&\textbf{(G.e)} && \mathrm{lock}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \mathrm{notify}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{success}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{token}_s^t \mathrel{\widetilde{\notin}} q \wedge \\
&&& \qquad \mathrm{ackU}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{go}_s^t \mathrel{\widetilde{\notin}} q \wedge \nexists u.\ \mathrm{newToken}_s^u \mathrel{\widetilde{\in}} q, \\
&\textbf{(H.s)} && \mathrm{ackU}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \exists u, L, W.\ \mathrm{waiting}_s(u, L, W) \mathrel{\widetilde{\in}} q \wedge u \neq t \wedge t \in W \wedge t \notin L, \\
&\textbf{(H.t)} && \mathrm{ackU}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \exists L, l, T.\ \mathrm{locking}_t(L, l, T) \mathrel{\widetilde{\in}} q \wedge s \notin T, \\
&\textbf{(H.e)} && \mathrm{ackU}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \mathrm{notify}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{success}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{loose}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{token}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{lock}_s^t \mathrel{\widetilde{\notin}} q \wedge \\
&&& \qquad \mathrm{ackL}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{unlock}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{go}_s^t \mathrel{\widetilde{\notin}} q \wedge \nexists u.\ \mathrm{newToken}_s^u \mathrel{\widetilde{\in}} q, \\
&\textbf{(I.s)} && \mathrm{ackL}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \exists u, L, W.\, \mathrm{waiting}_s(u, L, W) \mathrel{\widetilde{\in}} q \wedge u \neq t \wedge t{\in}W \wedge t \in L \vee \\
&&& \qquad \exists u, L, W.\, \mathrm{waiting}_s(u, L, W) \mathrel{\widetilde{\in}} q \wedge u \neq t \wedge t{\in}W \wedge t \notin L \wedge \mathrm{lock}_s^t \mathrel{\widetilde{\in}} q, \\
&\textbf{(I.t)} && \mathrm{ackL}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \exists L, l, T.\ \mathrm{locking}_t(L, l, T) \mathrel{\widetilde{\in}} q \wedge s \notin T, \\
&\textbf{(I.e)} && \mathrm{ackL}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \mathrm{notify}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{success}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{loose}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{token}_s^t \mathrel{\widetilde{\notin}} q \wedge \\
&&& \qquad \mathrm{ackU}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{unlock}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{go}_s^t \mathrel{\widetilde{\notin}} q \wedge \nexists u.\ \mathrm{newToken}_s^u \mathrel{\widetilde{\in}} q, \\
&\textbf{(J.s)} && \mathrm{unlock}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \exists L.\ \mathrm{locked}_s(t, L) \mathrel{\widetilde{\in}} q, \\
&\textbf{(J.t)} && \mathrm{unlock}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \exists L, l, T.\ \mathrm{locking}_t(L, l, T) \mathrel{\widetilde{\in}} q \wedge s \notin L \wedge l \neq s \vee \\
&&& \qquad \mathrm{lock}_s^t \mathrel{\widetilde{\in}} q, \\
&\textbf{(J.e)} && \mathrm{unlock}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \mathrm{notify}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{success}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{loose}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{token}_s^t \mathrel{\widetilde{\notin}} q \wedge \\
&&& \qquad \mathrm{ackU}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{ackL}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{go}_s^t \mathrel{\widetilde{\notin}} q \wedge \nexists u.\ \mathrm{newToken}_s^u \mathrel{\widetilde{\in}} q, \\
&\textbf{(K.s)} && \mathrm{go}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \exists L.\ \mathrm{locked}_s(t, L) \mathrel{\widetilde{\in}} q, \\
&\textbf{(K.t)} && \mathrm{go}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \exists T.\ \mathrm{firing}_t(T) \mathrel{\widetilde{\in}} q \wedge s \notin T, \\
&\textbf{(K.e)} && \mathrm{go}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \mathrm{notify}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{success}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{loose}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{token}_s^t \mathrel{\widetilde{\notin}} q \wedge \\
&&& \qquad \mathrm{lock}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{ackU}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{ackL}_s^t \mathrel{\widetilde{\notin}} q \wedge \mathrm{unlock}_s^t \mathrel{\widetilde{\notin}} q \wedge \nexists u.\ \mathrm{newToken}_s^u \mathrel{\widetilde{\in}} q, \\
&\textbf{(L.s)} && \mathrm{newToken}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \mathrm{empty}_s \mathrel{\widetilde{\in}} q, \\
&\textbf{(L.e)} && \mathrm{newToken}_s^t \mathrel{\widetilde{\in}} q \Rightarrow \nexists u, u \neq t.\ \mathrm{newToken}_s^u \mathrel{\widetilde{\in}} q, \\
&\textbf{(M.a)} && \mathrm{locking}_t(L, l, T) \mathrel{\widetilde{\in}} q \Rightarrow \forall s \in T.\ \mathrm{unlocked}_s \mathrel{\widetilde{\in}} q \vee \\
&&& \qquad \exists u, L'.\ \mathrm{locked}_s(u, L') \mathrel{\widetilde{\in}} q \vee \\
&&& \qquad \exists u, L', W.\ \mathrm{waiting}_s(u, L', W) \mathrel{\widetilde{\in}} q, \\
&\textbf{(M.b)} && \mathrm{locking}_t(L, l, T) \mathrel{\widetilde{\in}} q \Rightarrow \\
&&& \forall s \in T \setminus (L \cup \{l\}).\ \mathrm{unlocked}_s \mathrel{\widetilde{\in}} q \vee \\
&&& \qquad \exists u, L'.\ \mathrm{locked}_s(u, L') \mathrel{\widetilde{\in}} q \wedge u \neq t \wedge t \notin L' \vee \\
&&& \qquad \exists L'.\ \mathrm{locked}_s(t, L') \mathrel{\widetilde{\in}} q \wedge \mathrm{unlock}_s^t \mathrel{\widetilde{\in}} q \vee \\
&&& \qquad \exists u, L', W.\ \mathrm{waiting}_s(u, L', W) \mathrel{\widetilde{\in}} q \wedge u \neq t \wedge t \in W \wedge t \notin L, \\
&\textbf{(M.c)} && \mathrm{locking}_t(L, l, T) \mathrel{\widetilde{\in}} q \Rightarrow \forall s \in L \exists L'.\ \mathrm{locked}_s(t, L') \mathrel{\widetilde{\in}} q, \\
&\textbf{(M.d)} && \mathrm{locking}_t(L, l, T) \mathrel{\widetilde{\in}} q \wedge l \neq \bot \Rightarrow \exists u, L'.\ \mathrm{locked}_l(u, L') \mathrel{\widetilde{\in}} q \wedge t \neq u \wedge t \in L' \vee \\
&&& \qquad \mathrm{lock}_l^t \mathrel{\widetilde{\in}} q \vee \\
&&& \qquad \mathrm{success}_l^t \mathrel{\widetilde{\in}} q \vee \\
&&& \qquad \mathrm{loose}_l^t \mathrel{\widetilde{\in}} q,
\end{aligned}
$$

$$\begin{aligned}
&\textbf{(N.a)} && \text{firing}_t(T) \mathrel{\tilde{\in}} q \Rightarrow \forall s \in {}^\bullet t \setminus T.\ \exists L, W.\ \text{waiting}_s(t, L, W) \mathrel{\tilde{\in}} q \vee \\
&&& \qquad\qquad\qquad\qquad\qquad\qquad \text{go}^t_s \mathrel{\tilde{\in}} q \vee \\
&&& \qquad\qquad\qquad\qquad\qquad\qquad \text{token}^t_s \mathrel{\tilde{\in}} q, \\
&\textbf{(N.b)} && \text{firing}_t(T) \mathrel{\tilde{\in}} q \Rightarrow \forall s \in T.\ \text{empty}_s \mathrel{\tilde{\in}} q \wedge \nexists u.\ \text{newToken}^u_s \mathrel{\tilde{\in}} q, \\
&\textbf{(N.c)} && \text{firing}_t(T) \mathrel{\tilde{\in}} q \Rightarrow \forall s \in t^\bullet \setminus {}^\bullet t.\ \text{empty}_s \mathrel{\tilde{\in}} q \wedge \nexists u.\ \text{newToken}^u_s \mathrel{\tilde{\in}} q, \\
&\textbf{(N.d1)} && \text{firing}_t(T) \mathrel{\tilde{\in}} q \Rightarrow \forall s \in t^\bullet.\ \nexists u \in {}^\bullet s, u \neq t \exists T'.\ \text{firing}_u(T') \mathrel{\tilde{\in}} q, \\
&\textbf{(N.d2)} && \text{firing}_t(T) \mathrel{\tilde{\in}} q \Rightarrow \forall s \in {}^\bullet t.\ \nexists u \in s^\bullet, u \neq t \exists T'.\ \text{firing}_u(T') \mathrel{\tilde{\in}} q, \\
&\textbf{(N.d3)} && \text{firing}_t(T) \mathrel{\tilde{\in}} q \Rightarrow \forall s \in t^\bullet.\ \nexists u \in s^\bullet, u \neq t \exists T'.\ \text{firing}_u(T') \mathrel{\tilde{\in}} q, \\
&\textbf{(O.a)} && \text{waiting}_s(t, L, W) \mathrel{\tilde{\in}} q \Rightarrow \exists T.\ \text{firing}_t(T) \mathrel{\tilde{\in}} q \wedge s \notin T, \\
&\textbf{(O.b)} && \text{waiting}_s(t, L, W) \mathrel{\tilde{\in}} q \Rightarrow \forall u \in s^\bullet \setminus (W \cup \{t\}) \exists L', l, T.\ \text{locking}_u(L', l, T) \mathrel{\tilde{\in}} q \wedge s \notin T, \\
&\textbf{(O.c)} && \text{waiting}_s(t, L, W) \mathrel{\tilde{\in}} q \Rightarrow \forall u \in W.\ \text{loose}^u_s \mathrel{\tilde{\in}} q \vee \\
&&& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ackU}^u_s \mathrel{\tilde{\in}} q \vee \\
&&& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ackL}^u_s \mathrel{\tilde{\in}} q, \\
&\textbf{(P.a)} && \text{locked}_s(t, L) \mathrel{\tilde{\in}} q \Rightarrow \forall u \in L \exists L', T.\ \text{locking}_u(L', s, T) \mathrel{\tilde{\in}} q, \\
&\textbf{(P.b)} && \text{locked}_s(t, L) \mathrel{\tilde{\in}} q \Rightarrow \forall u \in s^\bullet \setminus L.\ \exists L', l, T.\ \text{locking}_u(L', l, T) \mathrel{\tilde{\in}} q \wedge s \in T \vee \\
&&& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{notify}^u_s \mathrel{\tilde{\in}} q \vee \\
&&& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{go}^u_s \mathrel{\tilde{\in}} q, \\
&\textbf{(P.c)} && \text{locked}_s(t, L) \mathrel{\tilde{\in}} q \Rightarrow \exists L', l, T.\ \text{locking}_t(L', l, T) \mathrel{\tilde{\in}} q \wedge s \in L' \vee \\
&&& \qquad\qquad\qquad\qquad\quad \text{success}^t_s \mathrel{\tilde{\in}} q \vee \\
&&& \qquad\qquad\qquad\qquad\quad \text{unlock}^t_s \mathrel{\tilde{\in}} q \vee \\
&&& \qquad\qquad\qquad\qquad\quad \text{go}^t_s \mathrel{\tilde{\in}} q, \\
&\textbf{(Q.a)} && \text{prenotify}_s \mathrel{\tilde{\in}} q \Rightarrow \forall u \in s^\bullet \exists L, l, T.\ \text{locking}_u(L, l, T) \mathrel{\tilde{\in}} q \wedge s \notin T, \text{and} \\
&\textbf{(R.a)} && \text{unlocked}_s \mathrel{\tilde{\in}} q \Rightarrow \forall u \in s^\bullet.\ \exists L, l, T.\ \text{locking}_u(L, l, T) \mathrel{\tilde{\in}} q \wedge s \in T \vee \\
&&& \qquad\qquad\qquad\qquad\qquad\qquad \text{notify}^u_s \mathrel{\tilde{\in}} q.
\end{aligned}$$

The invariant α could have been written more dense, but the presentation used here emphasises some properties of the terms which will be useful during the following proofs. First note that conditions (C.*) to (L.*), where the use of * means any character, all depend on the presence of some message, whereas conditions (M.*) to (R.*) depend on states.

Furthermore, most terms of the invariant deal just with the communication between a transition t and a place s without taking any other elements into account. Conditions (*.s) assert some properties of a place, conditions (*.t) assert properties of transitions and conditions (*.e) assert exclusiveness of messages.

The behavioural relation between the implementation and the original net is as follows: Whenever the implementation produces an output of fire^t, the original can fire the transition t, and similarly for sets of transitions as well.

Proposition 5.2.1

Let N be a plain net and let A_N be the FSM based implementation of it.

(i) $\mathfrak{f}(q_0^{A_N}) = M_0^N \wedge \mathfrak{f}'(q_0^{A_N}) = M_0^N$

(ii) $\alpha(q_0^{A_N})$

(iii) If $\alpha(q)$ and $q \xrightarrow{I;\emptyset}_{A_N} q'$ then $\mathfrak{f}(q) = \mathfrak{f}(q')$.

(iv) If $\alpha(q)$, $q \xrightarrow{I;O}_{A_N} q'$, and $O \neq \emptyset$ then $\mathfrak{f}(q) \xrightarrow{\{t \mid \text{fire}^t \in O\}}_N \mathfrak{f}(q')$.

(v) If $\alpha(q)$ and $q \xrightarrow{I;O}_{A_N} q'$ then $\alpha(q')$.

Proof

(i): No messages are travelling initially as per Definition 3.2.2. From Definition 5.2.1 follows that initially no transition t is in the state $\text{firing}_t(T)$ for any T. Furthermore from Definition 5.2.2 follows that every initially unmarked place s is in state empty_s and that every initially marked place s is in state prenotify_s. Thus $\mathfrak{f}(q_0^{A_N}) = M_0^N$ and $\mathfrak{f}'(q_0^{A_N}) = M_0^N$.

(ii): (A.*) by (i), (B) – (L.e) by the already noted fact that initially no messages are present. Every transition t is per Definition 5.2.1 initially in state $\text{locking}_t(\emptyset, \perp, \emptyset)$ thus $T = L = \emptyset$ and $l = \perp$ in (M.*) and all hold, as do (N.*). From Definition 5.2.2 follows that places are initially either in state empty_s or in state prenotify_s. Hence (O.*), (P.*) and (R.*), whereas (Q.a) follows from the fact that every transition t is in state $\text{locking}_t(\emptyset, \perp, \emptyset)$.

(iii): Due to Lemma 3.2.1 it suffices to show that the condition holds for singleton I. From Definition 3.2.2 follows that each singleton I must correspond to a step of a component FSM. The proof continues via case distinction over all such possible steps.

Case $\text{locking}_t(L, l, T) \xrightarrow{\{\text{notify}_s^t\};\emptyset}_{A_t} \text{locking}_t(L, l, T \cup \{s\})$, $s \notin T$: The consumption of notify_s^t didn't change $\mathfrak{f}$, neither did the state change of the transition.

Case $\text{locking}_t(L, l, T) \xrightarrow{\{\text{loose}_s^t\};\{\text{ackU}_s^t\}}_{A_t} \text{locking}_t(L, l, T \setminus \{s\})$, $s \in T, s \notin L, s \neq l \neq \perp$: The consumption of loose_s^t didn't change $\mathfrak{f}$, neither did the state change of the transition or the creation of ackU_s^t messages.

Case $\text{locking}_t(L, \perp, T) \xrightarrow{\{\text{loose}_s^t\};\{\text{ackU}_s^t\} \cup \{\text{unlock}_p^t \mid p \in L\}}_{A_t} \text{locking}_t(\emptyset, \perp, T \setminus \{s\})$, $s \in T \setminus L$: The consumption of loose_s^t didn't change $\mathfrak{f}$, neither did the state change of the transition or the creation of the new messages.

Case $\text{locking}_t(L, l, T) \xrightarrow{\{\text{loose}_l^t\};\{\text{ackL}_l^t\} \cup \{\text{unlock}_p^t \mid p \in L\}}_{A_t} \text{locking}_t(\emptyset, \perp, T \setminus \{l\})$: The consumption of loose_s^t didn't change $\mathfrak{f}$, neither did the state change of the transition or any of the produced messages.

Case $\text{locking}_t(L, \perp, {}^\bullet t) \xrightarrow{\{\text{internalLock}_l^t\};\{\text{lock}_l^t\}}_{A_t} \text{locking}_t(L, l, {}^\bullet t)$, $l = \min({}^\bullet t \setminus L)$: No message was consumed, lock_l^t messages don't affect $\mathfrak{f}$ and neither do the transition states.

Case $\text{locking}_t(L, l, {}^\bullet t) \xrightarrow{\{\text{success}_l^t\};\emptyset}_{A_t} \text{locking}_t(L \cup \{l\}, \perp, {}^\bullet t)$: Again, success_l^t messages don't affect $\mathfrak{f}$ and neither do the $\text{locking}_t(\ldots)$ states.

Case $\mathrm{locking}_t(L,l,T) \xrightarrow{\{\mathrm{success}^t_l\};\left\{\mathrm{unlock}^t_p \;\middle|\; p\in L\cup\{l\}\right\}}_{A_t} \mathrm{locking}_t(\emptyset,\bot,T)$, $T \neq {}^\bullet t$: Basically as above.

Case $\mathrm{locking}_t({}^\bullet t,\bot,{}^\bullet t) \xrightarrow{\{\mathrm{internalFire}^t\};\{\mathrm{fire}^t\}\cup\left\{\mathrm{go}^t_s \;\middle|\; s\in{}^\bullet t\right\}}_{A_t} \mathrm{firing}_t(\emptyset)$: This step is not possible as the fire^t action is not an input of any other component and is thus visible in the outside step, violating the assumption that the step has no observable output.

Case $\mathrm{firing}_t(T) \xrightarrow{\{\mathrm{token}^t_s\};\emptyset}_{A_t} \mathrm{firing}_t(T\cup\{s\})$, $s \notin T$: The token^t_s message does not affect $\mathfrak{f}$ and neither do the contents of T, as long as the transition stays in a state of $\mathrm{firing}_t(\ldots)$.

Case $\mathrm{firing}_t({}^\bullet t) \xrightarrow{\{\mathrm{internalDone}^t\};\left\{\mathrm{newToken}^t_s \;\middle|\; s\in t^\bullet\right\}}_{A_t} \mathrm{locking}_t(\emptyset,\bot,\emptyset)$: For all $s \in t^\bullet$, it might be the case that no transition $u \in {}^\bullet s$ in state $\mathrm{firing}_u(\ldots)$ exists any more, but a $\mathrm{newToken}^t_s$ message has been created for exactly those places. From $\alpha(q)$ (N.b), (N.c) and (K.s) follows that no go^u_s messages are currently travelling towards any postplaces of t.

Case $\mathrm{empty}_s \xrightarrow{\{\mathrm{newToken}^t_s\};\emptyset}_{A_s} \mathrm{prenotify}_s$, $t \in {}^\bullet s$: While the $\mathrm{newToken}^t_s$ message has been consumed, the state of s changed to $\mathrm{prenotify}_s$ thus preserving $\mathfrak{f}$.

Case $\mathrm{prenotify}_s \xrightarrow{\{\mathrm{internalNotify}^s\};\left\{\mathrm{notify}^t_s \;\middle|\; t\in s^\bullet\right\}}_{A_s} \mathrm{unlocked}_s$: The place s contributes to $\mathfrak{f}$ whether it is in state $\mathrm{prenotify}_s$ or in state $\mathrm{unlocked}_s$. The messages produced don't affect $\mathfrak{f}$.

Case $\mathrm{unlocked}_s \xrightarrow{\{\mathrm{lock}^t_s\};\{\mathrm{success}^t_s\}}_{A_s} \mathrm{locked}_s(t,\emptyset)$: The place s contributes to $\mathfrak{f}$ whether it is in state $\mathrm{unlocked}_s$ or in some state $\mathrm{locked}_s(\ldots)$. The messages lock^t_s and $\mathrm{success}^t_s$ don't affect $\mathfrak{f}$.

Case $\mathrm{locked}_s(t,L) \xrightarrow{\{\mathrm{lock}^u_s\};\emptyset}_{A_s} \mathrm{locked}_s(t,L\cup\{u\})$, $u \neq t, u \notin L$: As long as the place s stays in some state $\mathrm{locked}_s(\ldots)$ it contributes to $\mathfrak{f}$. The message consumed doesn't affect $\mathfrak{f}$.

Case $\mathrm{locked}_s(t,L) \xrightarrow{\{\mathrm{unlock}^t_s\};\{\mathrm{success}^u_s\}}_{A_s} \mathrm{locked}_s(u,L\setminus\{u\})$, $u \in L$: As long as the place s stays in some state $\mathrm{locked}_s(\ldots)$ it contributes to $\mathfrak{f}$. The messages unlock^t_s and $\mathrm{success}^u_s$ don't affect $\mathfrak{f}$.

Case $\mathrm{locked}_s(t,\emptyset) \xrightarrow{\{\mathrm{unlock}^t_s\};\emptyset}_{A_s} \mathrm{unlocked}_s$: The place s contributes to $\mathfrak{f}$ whether it is in state $\mathrm{locked}_s(t,\emptyset)$ or in $\mathrm{unlocked}_s$. The unlock^t_s message doesn't affect $\mathfrak{f}$.

Case $\mathrm{locked}_s(t,L) \xrightarrow{\{\mathrm{go}^t_s\};\{\mathrm{loose}^u_s \;\mid\; u\in s^\bullet, u\neq t\}}_{A_s} \mathrm{waiting}_s(t,L,s^\bullet\setminus\{t\})$: The state of place s does not contribute to $\mathfrak{f}$ after this step, but it did not before either, due to the presence of the go^t_s message.

Case $\mathrm{waiting}_s(t,L,W) \xrightarrow{\{\mathrm{lock}^u_s\};\emptyset}_{A_s} \mathrm{waiting}_s(t,L\cup\{u\},W)$, $u \notin L, u \in W$: The state of the place does not contribute to $\mathfrak{f}$ in any state $\mathrm{waiting}_s(\ldots)$, neither does the lock^u_s message.

Case $\mathrm{waiting}_s(t,L,W) \xrightarrow{\{\mathrm{ackL}^u_s\};\emptyset}_{A_s} \mathrm{waiting}_s(t,L\setminus\{u\},W\setminus\{u\})$, $u \in L$: The state of the place does not contribute to $\mathfrak{f}$ in any state $\mathrm{waiting}_s(\ldots)$, neither does the ackL^u_s message.

Case $\mathrm{waiting}_s(t,L,W) \xrightarrow{\{\mathrm{ackU}^u_s\};\emptyset}_{A_s} \mathrm{waiting}_s(t,L,W\setminus\{u\})$, $u \notin L, u \in W$: The state of the place does not contribute to $\mathfrak{f}$ in any state $\mathrm{waiting}_s(\ldots)$, neither does the ackU^u_s message.

Case $\mathrm{waiting}_s(t, \emptyset, \emptyset) \xrightarrow{\{\mathrm{internalPassToken}^t_s\};\{\mathrm{token}^t_s\}}_{A_s} \mathrm{empty}_s$: The state of the place does not contribute to $\mathfrak{f}$, neither in state $\mathrm{waiting}_s(t, \emptyset)$ nor in state empty_s. The message token^t_s does not change $\mathfrak{f}$.

(iv): As before, only singleton I need to be considered. From Definition 3.2.2, Definition 5.2.1 and Definition 5.2.2 follows that the only visible outputs are of the form fire^t. Thus the only possible step is $\mathrm{locking}_t({}^\bullet t, \bot, {}^\bullet t) \xrightarrow{\{\mathrm{internalFire}^t\};\{\mathrm{fire}^t\}\cup\left\{\mathrm{go}^t_s \;\middle|\; s\in{}^\bullet t\right\}}_{A_t} \mathrm{firing}_t(\emptyset)$.

As N is assumed contact free, it suffices to show that ${}^\bullet t \subseteq \mathfrak{f}(q)$ and $\mathfrak{f}(q') = (\mathfrak{f}(q) \setminus {}^\bullet t) \cup t^\bullet$.

From $\alpha(q)$ (M.c) follows that every preplace s of t is in some state $\mathrm{locked}_s(t, \ldots)$. From (K.t) follows that no go^t_s message is travelling, as t is not in any state $\mathrm{firing}_t(\ldots)$ in q. Thus every preplace of t is in $\mathfrak{f}(q)$.

For every preplace s of t one message go^t_s is produced, effectively removing s from $\mathfrak{f}(q')$ unless s is also a postplace of t, which is now in state $\mathrm{firing}_t(\emptyset)$. That s does not remain in $\mathfrak{f}(q')$ due to some concurrently firing transition u which also has s in its postset follows from $\alpha(q)$ (M.c) (every preplace s of t is in a state $\mathrm{locked}_s(t, \ldots)$), (N.c) (postplaces p of u which are not in ${}^\bullet u$ are in state empty_p), (N.b) and (N.a) (preplaces p of u are either in state empty_p or in a state $\mathrm{waiting}_p(\ldots)$ or a go^u_p or a token^u_p message is travelling) and (F.s) and (K.s) (either message is incompatible with the fact that s is locked to t).

Thus $\mathfrak{f}(q') = (\mathfrak{f}(q) \setminus {}^\bullet t) \cup t^\bullet$.

(v): (A.a) from (iii) and (iv).

Some parts of (C.e) can be proven from the rest of the invariant. No $\mathrm{success}^t_s$ can exist as (C.t) and (D.t). No token^t_s can exist as (C.t) and (F.t). No unlock^t_s can exist as (C.s) and (J.s). No go^t_s can exist as (C.s) and (K.s). No $\mathrm{newToken}^u_s$ can exist as (C.s) and (L.s). Thus I will instead of (C.e) show $\mathrm{notify}^t_s \mathrel{\widetilde{\in}} q \Rightarrow \mathrm{lock}^t_s \mathrel{\widetilde{\notin}} q \wedge \mathrm{ackU}^t_s \mathrel{\widetilde{\notin}} q \wedge \mathrm{ackL}^t_s \mathrel{\widetilde{\notin}} q$.

Similarly for (D.e) via the following deductions. No notify^t_s can exist as (C.e). No loose^t_s can exist as (D.s) and (E.s). No token^t_s can exist as (D.s) and (F.s). No ackU^t_s can exist as (D.s) and (H.s). No ackL^t_s can exist as (D.s) and (I.s). No go^t_s can exist as (D.t) and (K.t). No $\mathrm{newToken}^u_s$ can exist as (D.s) and (L.s). Assume now that lock^t_s exists. Then from (D.s) and (G.s) follows that also unlock^t_s exists. Assume that unlock^t_s exists. Then from (D.t) and (J.t) follows that also lock^t_s exists. Thus I will instead of (D.e) show $\mathrm{success}^t_s \mathrel{\widetilde{\in}} q \Rightarrow \mathrm{lock}^t_s \mathrel{\widetilde{\notin}} q \vee \mathrm{unlock}^t_s \mathrel{\widetilde{\notin}} q$.

Repeating the same for (E.e). No $\mathrm{success}^t_s$ can exist as (D.e). No token^t_s can exist as (E.s) and (F.s). No ackU^t_s can exist as (E.t), (C.e) and (H.t). No ackL^t_s can exist as (E.t), (C.e) and (I.t). No unlock^t_s can exist as (E.s) and (J.s). No go^t_s can exist as (E.s) and (K.s). No $\mathrm{newToken}^u_s$ can exist as (E.s) and (L.s). Thus (E.e).

Repeating the same for (F.e). No notify^t_s can exist as (C.e). No $\mathrm{success}^t_s$ can exist as (D.e). No loose^t_s can exist as (E.e). No lock^t_s can exist as (F.s) and (G.s). No ackU^t_s can exist as (F.s) and (H.s). No ackL^t_s can exist as (F.s) and (I.s). No unlock^t_s can exist as (F.s) and (J.s). No go^t_s can exist as (F.s) and (K.s). Thus I will instead of (F.e) show $\mathrm{token}^t_s \mathrel{\widetilde{\in}} q \Rightarrow \nexists u.\ \mathrm{newToken}^u_s \mathrel{\widetilde{\in}} q$.

Repeating the same for (G.e). No notify_s^t can exist as (C.e). No success_s^t can exist as (D.e). No token_s^t can exist as (F.e). No go_s^t can exist as (G.t), (I.t), and (K.t). No newToken_s^u can exist as (G.s) and (L.s). Thus I will instead of (G.e) show $\text{lock}_s^t \mathrel{\tilde{\in}} q \Rightarrow \text{ackU}_s^t \mathrel{\tilde{\notin}} q$.

Repeating the same for (H.e). No notify_s^t can exist as (C.e). No success_s^t can exist as (D.e). No loose_s^t can exist as (E.e). No token_s^t can exist as (F.e). No lock_s^t can exist as (G.e). No unlock_s^t can exist as (H.s) and (J.s). No go_s^t can exist as (H.s) and (K.s). No newToken_s^u can exist as (H.s) and (L.s). Thus I will instead of (H.e) show $\text{ackU}_s^t \mathrel{\tilde{\in}} q \Rightarrow \text{ackL}_s^t \mathrel{\tilde{\notin}} q$.

Repeating the same for (I.e). No notify_s^t can exist as (C.e). No success_s^t can exist as (D.e). No loose_s^t can exist as (E.e). No token_s^t can exist as (F.e). No ackU_s^t can exist as (H.e). No unlock_s^t can exist as (I.s) and (J.s). No go_s^t can exist as (I.s) and (K.s). No newToken_s^u can exist as (I.s) and (L.s). Thus (I.e).

Repeating the same for (J.e). No notify_s^t can exist as (C.e). No success_s^t can exist as (D.e). No loose_s^t can exist as (E.e). No token_s^t can exist as (F.e). No ackU_s^t can exist as (H.e). No ackL_s^t can exist as (I.e). No go_s^t can exist as (J.t), (K.t), (G.t), and (I.t). No newToken_s^u can exist as (J.s) and (L.s). Thus (J.e).

Repeating the same for (K.e). No notify_s^t can exist as (C.e). No success_s^t can exist as (D.e). No loose_s^t can exist as (E.e). No token_s^t can exist as (F.e). No lock_s^t can exist as (G.e). No ackU_s^t can exist as (H.e). No ackL_s^t can exist as (I.e). No unlock_s^t can exist as (J.e). No newToken_s^u can exist as (K.s) and (L.s). Thus (K.e).

Due to Lemma 3.2.1 it suffices to show that the other conditions holds for singleton I. From Definition 3.2.2 follows that each singleton I must correspond to a step of a component FSM. The proof continues via case distinction over all such possible steps. The attentive reader might suspect now that a case distinction over many cases, each proving quite a lot of invariant terms, is rather tedious. It is indeed quite a lot of work, so whoever finds it too lengthy is suggested to skip the rest of this proof.

While referring to the clauses of Definition 5.2.5, the following uses (X) to denote the respective clause of $\alpha(q)$ and (X)' to denote clauses from $\alpha(q')$.

Case $\text{locking}_t(L, l, T) \xrightarrow{\{\text{notify}_s^t\};\emptyset}_{A_t} \text{locking}_t(L, l, T \cup \{s\})$, $s \notin T$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' as no messages are produced. (*.s)' as no state of a place implementation is modified, no new message was generated, (G.s)' asserts the existence of an unlock_s^t message, (I.s)' asserts the existence of a lock_s^t message, and neither was consumed. (*.e)' as no new messages have been produced.

(C.t)' the only value added to T is s and only one notify_s^t message existed in q as per (B). (D.t)' and (G.t)' with the two existing values L and $T \cup \{s\}$ and the fact that no ackL_p^u message was consumed. (E.t)' as the only notify_p^t message consumed has $p = s$, s was added to T and $s \notin L$. (F.t)' from (F.t). (H.t)' from (C.e) as only s was added, and no ackU_s^t message can exist. (I.t)' with the same argument for ackL_s^t. (J.t)' as nothing relevant changed from (J.t). And (K.t)' from (K.t).

(M.a)' and (M.b)' from (C.s), (M.c)' from the fact that L stayed unchanged. (M.d)' as no relevant messages have been consumed and l didn't change. (N.*)' and (O.a)' as no terms therein have changed. (O.b)' from (C.s) since if s is in some state $\text{waiting}_s(u, L, W)$ then $t \in W$ and u in (O.b)' does not range over t. No terms in (O.c)' and (P.a)' have changed, and (P.b)' stays true as well, as while the notify_s^t message has been consumed, s was added to T. (P.c)' as no relevant messages have been consumed and only T was changed. (Q.a)' from (C.s) and (R.a)' with the same argument as (P.b)'.

Case $\text{locking}_t(L, l, T) \xrightarrow{\{\text{loose}_s^t\};\{\text{ackU}_s^t\}}_{A_t} \text{locking}_t(L, l, T \setminus \{s\})$, $s \in T, s \notin L, s \neq l \neq \bot$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' from (E.e).

(C.s)' as no place state was changed. No notify_s^t message existed per $s \in T$ and (C.t). Thus (C.t)' and (C.e)'.

(D.s)' as no place state was changed. (D.t)' as only T was changed. (D.e)' from (E.e).

(E.s)' as no place state was changed. (E.t)' as only s was removed from T, L remained equal, no second loose_s^t message existed as per (B), and no notify_p^u message was consumed. (E.e)' from (B).

(F.s)' as no place state was changed. (F.t)' from (F.t). (F.e)' from (E.e).

(G.s)' as no place state was changed. (G.t)' as only T was changed and no ackL_p^u message was consumed. (G.e)' as with $s \neq l$ no lock_s^t message can exists per (G.t) and (E.e).

(H.s)' from (E.s) and (M.b). (H.t)' trivially from the performed step. (H.e)' from (E.e) which enforces that no ackL_s^t message can exist.

(I.s)' as no place state was changed and no lock_p^u was consumed. (I.t)' as something was removed from T.

(J.s)' as no place state was changed. (J.t)' as only T was changed and no lock_p^u was consumed.

(K.s)' as no place state was changed. (K.t)' from (K.t).

(L.s)' as no place state was modified. (L.e)' as no newToken_p^u messages were produced.

Terms only improved for (M.a)', (M.b)', (M.c)', (N.*)', (O.a)', (O.b)', (P.a)', (P.c)', and (Q.a)'. (M.d)' as the consumed loose_s^t message has $s \neq l$. (O.c)' as the loose_s^t was replaced by the ackU_s^t message. Note that s is in a state $\text{waiting}_s(\dots)$ from (E.s). Thus (P.b)' and (R.a)'.

Case $\text{locking}_t(L, \bot, T) \xrightarrow{\{\text{loose}_s^t\};\{\text{ackU}_s^t\}\cup\left\{\text{unlock}_p^t \;\middle|\; p\in L\right\}}_{A_t} \text{locking}_t(\emptyset, \bot, T \setminus \{s\})$, $s \in T \setminus L$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' as an earlier ackU_s^t message is excluded per (E.e) and the unlock_p^t are unproblematic as per (J.t), (G.t), and (I.t).

(C.s)' as no place state was changed. (C.t)' as T became smaller. As the only critical message for (C.e)' is the ackU_s^t message, it suffices that from (C.t) follows that no notify_s^t message existed in q.

From (D.t) follows that no success^t_p message can exist in q. Thus (D.*)'.

(E.s)' as no place state was changed. (E.t)' as the only element removed from T was s. There existed only one loose^t_s message per (B) and that was consumed.

From (F.s) follows that no token^u_s message existed before. Thus (F.*)'.

(G.s)' as no place state was changed and no unlock^u_r message was consumed. Assume some $\text{lock}^t_r \,\widetilde{\in}\, q$. Then per (G.t) there must also exist some $\text{ackL}^t_r \,\widetilde{\in}\, q$, which was not consumed. Thus (G.t)'. From (I.t), no such ackL^t_r message can exist for any $s \in T$ however, hence $\text{lock}^t_s \,\widetilde{\notin}\, q$ and thus (G.e)'.

(H.s)' from (E.s) and (M.b). (H.t)' trivially from the performed step. (H.e)' from (E.e) which enforces that no ackL^t_s message can exist.

(I.s)' as no place state was changed and no lock^u_r was consumed. (I.t)' as something was removed from T.

(J.s)' as no place state was changed. (J.t)' as L became smaller and no lock^u_r was consumed.

From (K.t) follows that no go^t_r message can exist. Thus (K.*)'. (L.s)' as no place state was modified. (L.e)' as no newToken^u_r messages were produced.

Terms only improved for (M.a)', (M.c)', (N.*)', (O.a)', (O.b)', (P.a)', and (Q.a)'.

(M.b)' from (M.c) and the newly produced unlock^t_p messages. (M.d)' as the only loose^u_r message consumed has $r = s$ and $u = t$, but t is in state $\text{locking}_t(\emptyset, \bot, T \setminus \{s\})$ after the step. (O.c)' as the loose^t_s message was replaced by the newly produced ackU^t_s message. Note that s is in a state $\text{waiting}_s(\ldots)$ from (E.s). Thus (P.b)'. (P.c)' with the newly produced unlock^t_p messages. (R.a)' with the same argument as (P.b)'.

Case $\text{locking}_t(L, l, T) \xrightarrow{\{\text{loose}^t_l\};\{\text{ackL}^t_l\}\cup\left\{\text{unlock}^t_p \;\middle|\; p\in L\right\}}_{A_t} \text{locking}_t(\emptyset, \bot, T \setminus \{l\})$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' from (E.e) and (J.t), (G.t), and (I.t).

(C.s)' as no place state was changed. (C.t)' as T became smaller. From (C.t) with the performed step follows that no notify^t_r message existed for $r = l$. Thus (C.e)'.

(D.s)' as no place state was changed. (D.t)' as only success^t_r messages with $r = l$ are possible from (D.t) but (E.e) and thus no such message exists. Thus also (D.e)'.

(E.s)' as no place state was changed. (E.t)' as the only element removed from T was l. The only problematic message is thus loose^t_l which was consumed however and existed only once as per (B). Also no notify^u_r message was consumed.

From (F.t) no messages token^t_r can exist. Thus (F.*)'.

(G.s)' as no place state was changed and no unlock^u_r message was consumed. (G.t)' as for a possible $\text{lock}^t_l \,\widetilde{\in}\, q$ there is $\text{ackL}^t_l \,\widetilde{\in}\, q'$ and for some $\text{lock}^t_r \,\widetilde{\in}\, q$ with $r \in L$ there must be an ackL_r^t message already as per (G.t). Thus (G.t)'. (G.e)' as neither lock_r^u nor ackU_r^u messages have been produced.

(H.s)' as no place state was changed. (H.t)' as T became smaller. (H.e)' as the only new ackL_r^u message has $r = l$ and $u = t$ and (E.e).

(I.s)' as no place state was changed and no lock_r^u was consumed. (I.t)' as T became smaller and l was specifically removed. (J.s)' as no place state was changed. (J.t)' as no lock_r^u message was consumed and no place equals $\perp$ or is in the empty set.

From (K.t) follows that no fire_r^t message existed, thus (K.*)'. (L.s)' as no place state was modified. (L.e)' as no $\mathrm{newToken}_r^u$ messages were produced.

Terms only improved for (M.a)' (M.c)' (N.*)' (O.a)', (O.b)', and (Q.a)'. (M.b)' as for all $s \in L$ (M.c) implies that $\mathrm{locked}_s(t, L') \mathrel{\tilde{\in}} q$ for some L' and the step generated respective unlock_s^t messages. (M.d)' as the only message consumed was loose_l^t and in q' the transition t is in the state $\mathrm{locking}_t(\emptyset, \perp, T \setminus \{l\})$ which is unproblematic for (M.d)'. (O.c)' as the loose_l^t message was replaced by ackL_l^t. Per (P.a) t was only in one L of a $\mathrm{locked}_r(u, L) \mathrel{\tilde{\in}} q$, namely with $r = l$. From (E.s) however, that state is no longer present. Thus (P.a)' and with the fact that only l was removed from T also (P.b)'. (P.c)' with the newly produced unlock_p^t messages. From (P.a), (E.s), and that only l was removed also (R.a)'.

Case $\mathrm{locking}_t(L, \perp, {}^\bullet t) \xrightarrow{\{\mathrm{internalLock}_l^t\};\{\mathrm{lock}_l^t\}}_{A_t} \mathrm{locking}_t(L, l, {}^\bullet t)$, $l = \min({}^\bullet t \setminus L)$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' from (G.t) and (I.t).

From (C.t) follows that $\mathrm{notify}_l^t \mathrel{\tilde{\notin}} q$. Thus (C.*)'. From (D.t) follows that no $\mathrm{success}_p^t$ message can exist in q. Thus (D.*)'.

(E.s)' as no place state was changed. (E.t)' as no notify_p^u messages were consumed and the first and last components of the transition state didn't change.

From (F.t) no messages token_p^t can exist. Thus (F.*)'. With (H.t) for (H.*)'. With (I.t) for (I.*)'. With (K.t) for (K.*)'.

The above argument with (G.t) and (I.t) works towards (G.*)' for all messages but the newly produced lock_l^t. Still (G.s)' together with (M.b), (G.t)' from the step, (G.e)' from the fact that no ackU_l^t message exists per (H.t).

(J.s)' as no place state was changed. Assume there existed some $\mathrm{unlock}_p^t \mathrel{\tilde{\in}} q$. If $p \neq l$ everything stays well, if $p = l$ then the appropriate lock_p^t was produced, thus (J.t)'.

(L.s)' as no place state was modified. (L.e)' as no $\mathrm{newToken}_p^u$ messages were produced.

Terms only improved for (M.a)', (M.b)', (M.c)', (N.*)', (O.*)', (P.*)', (Q.*)' and (R.*)'. (M.d)' with the newly produced lock_l^t message.

Case $\mathrm{locking}_t(L, l, {}^\bullet t) \xrightarrow{\{\mathrm{success}_l^t\};\emptyset}_{A_t} \mathrm{locking}_t(L \cup \{l\}, \perp, {}^\bullet t)$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' as no messages are produced. From (C.t) follows that no notify_p^t message can exist. Thus (C.*)'.

From (B) and (D.t) follows that exactly one success_p^t message can exist, which has $p = l$. It was consumed though, so (D.*)'.

(E.s)' as no place state was changed. (E.t)' as no notify_p^u messages were consumed, the last component of the transition state didn't change, and the only element added to L was l for which (D.e) guarantees that no loose_l^t message exists.

From (F.t) follows that no fire_p^t message exists. Thus (F.*)'.

From (G.t) and (I.t) follows that every lock_p^t message must have $p = l$. By (D.e) no such message exists and (G.*)'.

From (H.t) follows that no ackU_p^t message exists and (H.*)'. Using (I.t), (I.*)' follows similarly.

(J.s)' as no place state was changed. (J.t)' as the only element added to L was l.

(K.*)' again via (K.t). (L.s)' as no place state was modified. (L.e)' as no newToken_p^u messages were produced.

Terms only improved for (M.a)', (M.b)', (N.*)', (O.*)', (P.b)', (Q.a)', and (R.a)'.

(M.c)' with (D.s). (M.d)' as the only message consumed was success_l^t and in q' the transition t is in the state $\text{locking}_t(L \cup \{l\}, \bot, {}^\bullet t)$ which is unproblematic for (M.d)'. (P.a)' as from (D.s) follows that l is in a state $\text{locked}_l(t, L')$ with $t \notin L'$ per Definition 5.2.2. (P.c)' as only the success_l^t message was removed and l was added to L.

Case $\text{locking}_t(L, l, T) \xrightarrow{\{\text{success}_l^t\};\{\text{unlock}_p^t \mid p \in L \cup \{l\}\}}_{A_t} \text{locking}_t(\emptyset, \bot, T)$, $T \neq {}^\bullet t$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. Assume some unlock_p^t message already existed with $p = l$ or $p \in L$. If $p = l$ there is a contradiction with (D.e), hence $p \in L$. For $p \in L$ however (J.t), (G.t) and then (I.t) constitute a contradiction as well. So no such unlock_p^t message existed and (B)'.

(C.s)' as no place state was modified. (C.t)' as T remained equal. (C.e)' as no lock_r^u, ackU_r^u, or ackL_r^u messages have been produced.

From (D.t) and (B) follows that no further success_r^t message existed. Thus (D.*)'.

(E.s)' as no place state was changed. (E.t)' as no notify_r^u messages were consumed and the last component of the transition state didn't change.

From (F.t) follows that no fire_r^t message exists. Thus (F.*)'. From (K.t) similarly (K.*)'.

(G.s)' as no place state was modified and no unlock_r^u message was consumed. (G.t)' as from (D.e) no lock_l^t existed and for all other $\text{lock}_r^t \mathrel{\widetilde{\in}} q$ (G.t) guarantees that there is an $\text{ackL}_r^t \mathrel{\widetilde{\in}} q$ which was not consumed. (G.e)' as neither lock_r^u nor ackU_r^u messages have been produced.

(H.s)' as no place state was modified. (H.t)' as T remained equal. (H.e)' as neither ackU_r^u nor ackL_r^u messages have been created.

(I.s)' as no place state was modified and no lock_r^u have been consumed. (I.t)' as T remained equal.

As argued for (B)' no unlock_p^t messages existed before the step. Now however, unlock_p^t messages exist, one with $p = l$ and the others with $p \in L$. For the one with $p = l$ (J.s)' follows from (D.s). For those with $p \in L$ (J.s)' from (M.c). (J.t)' from the performed step.

(L.s)' as no place state was modified. (L.e)' as no newToken_p^u messages were produced.

Terms only improved for (M.a)', (M.c)', (N.*)', (O.*)', (P.b)', (Q.a)', and (R.a)'.

(M.b)' from (D.s), (M.c), and the newly produced unlock_p^t messages. (M.d)' as the only message consumed was success_l^t and in q' the transition t is in the state $\text{locking}_t(\emptyset, \bot, T)$ which is unproblematic for (M.d)'.

Assume a place p existed in state $\text{locked}_p(u, L)$ with $t \in L$. Then $p = l$ from (P.a). Then there is a contradiction with (D.s). Thus no such place exists and (P.a)'. (P.c)' as the success_l^t message was replaced by an unlock_l^t message.

Case $\text{locking}_t({}^\bullet t, \bot, {}^\bullet t) \xrightarrow{\{\text{internalFire}^t\};\{\text{fire}^t\}\cup\left\{\text{go}_s^t \;\mid\; s\in{}^\bullet t\right\}}_{A_t} \text{firing}_t(\emptyset)$:

Then $\alpha(q')$ as follows: (A.b)' as all preplaces s of t are currently in a state $\text{locked}_s(t, L)$ for some L per (M.c). Thus $\mathfrak{f}'$ didn't change. The fire^t message is an output of the composed state machine and does not affect (B)'. From (K.t) no go_p^t message existed before the step, thus (B)'.

From (C.t) no notify_p^t message existed, thus (C.*)'. From (D.t) similarly (D.*)'. From (E.s) and (M.c) thus (E.*)'. From (F.t) thus (F.*)'. From (G.t) and (I.t) similarly (G.*)'. From (H.t) thus (H.*)'. From (I.t) thus (I.*)'. From (J.t), (G.t), and (I.t) thus (J.*)'.

(K.s)' from (M.c). (K.t)' trivially from the performed step.

(L.s)' as no place state was modified. (L.e)' as no newToken_p^u messages were produced.

Terms only improved for (M.*)', (O.a)', and (O.c)'.

(N.a)' from the produced go_s^t messages. (N.b)' as T is empty after the step.

From (M.c) follows that every place s in ${}^\bullet t$ is in state $\text{locking}_s(t, L)$ with some L. From (K.t) no go_s^t message existed before the step.

From (A.b) and Definition 5.2.4 then ${}^\bullet t \subseteq \mathfrak{f}'(q)$. As N was assumed to be contact free, then for every place s in $t^\bullet \setminus {}^\bullet t$, $s \notin \mathfrak{f}'(q)$. Thus s must be in state empty_s and no newToken_s^u message exists. Thus (N.c)'.

Also from (A.b) and Definition 5.2.4, ${}^\bullet t \subseteq \mathfrak{f}(q)$. As N was assumed to be contact free, then for every place s in $t^\bullet \setminus {}^\bullet t$, $s \notin \mathfrak{f}(q)$. Thus there cannot exist $u \in {}^\bullet s$ with $\text{firing}_u(T') \mathrel{\tilde{\in}} q$ for some T'. Hence (N.d1)'.

Assume some $u \neq t$ with $\text{firing}_u(U) \mathrel{\tilde{\in}} q$ for some U and $p \in {}^\bullet t \cap {}^\bullet u$ existed. Then per (M.c) and (N.b) $p \notin U$. With (M.c), (N.a), and (K.s) then $\text{token}_p^u \mathrel{\tilde{\in}} q$. But then (F.s) and (M.c) form a contradiction. Thus no such u can exist and (N.d2)'.

As already argued for (N.c)', for every $s \in t^\bullet \setminus {}^\bullet t$, $s \notin \mathfrak{f}'(q)$ and per Definition 5.2.4 no $u \in s^\bullet$ with $\text{firing}_u(\ldots) \mathrel{\tilde{\in}} q$ can exist. For $s \in {}^\bullet t$ the same arguments as for (N.d2)' can be applied, again showing that no $u \in s^\bullet$ with $\text{firing}_u(\ldots) \mathrel{\tilde{\in}} q$ exists. Thus no such u exists for any $s \in t^\bullet$ and (N.d3)'.

Assume there existed some place p with $\text{waiting}_p(u, L, W) \mathrel{\tilde{\in}} q$ and $t \in p^\bullet \setminus (W \cup \{u\})$. Then there would be a contradiction between (O.b) and the initial state of the step. Thus no such place exists and (O.b)'. Using (P.a) a similar argument shows (P.a)'.

(P.b)' and (P.c)' with the produced go_s^t messages. (Q.a)' and (R.a)' as all preplaces p of t are in a state $\text{locked}_p(t, L)$ for some L per (M.c).

Case $\text{firing}_t(T) \xrightarrow{\{\text{token}_s^t\};\emptyset}_{A_t} \text{firing}_t(T \cup \{s\})$, $s \notin T$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' as no messages are produced. Thus also (*.e)'.

From (C.t) no notify_p^t message existed, thus (C.*)'. Similarly (D.t) shows (D.*)'. (E.t) and (C.t) thus (E.*)'. (G.t) and (I.t) thus (G.*)'. (H.t) thus (H.*)'. (I.t) thus (I.*)'. (J.t), (G.t), and (I.t) thus (J.*)'.

(F.s)' as no place state was changed. (F.t)' as the only place added to T was s and via (B) no second token_s^t message existed.

(K.s)' as no place state was changed. (K.t)' as the only place added to T was s and via (F.e) no go_s^t message existed.

(L.s)' as no place state was modified. (L.e)' as no newToken_p^u messages were produced.

Terms only improved for (M.*)', (N.c)', (N.d1)', (N.d2)', (N.d3)', (O.b)', (O.c)', (P.*)', (Q,a)' and (R.a)'.

(N.a)' as the only message consumed was token_s^t and s was added to T. (N.b)' from (F.s) and (F.e). (O.a)' as the only place added to T was s and (F.s) enforces that s is in state empty_s.

Case $\text{firing}_t({}^\bullet t) \xrightarrow{\{\text{internalDone}^t\};\left\{\text{newToken}_s^t \;\middle|\; s \in t^\bullet\right\}}_{A_t} \text{locking}_t(\emptyset, \perp, \emptyset)$:

Then $\alpha(q')$ as follows: From (A.b) and Definition 5.2.4 follows that ${}^\bullet t \subseteq \mathfrak{f}'(q')$. As N was assumed to be contact free, thus $\mathfrak{f}'(q)\ [\{t\}\rangle_N\ (\mathfrak{f}'(q) \setminus {}^\bullet t) \cup t^\bullet$. With the performed step and (N.d2) follows that $\mathfrak{f}'(q') = (\mathfrak{f}'(q) \setminus {}^\bullet t) \cup t^\bullet$. Thus (A.b)'.

For every $s \in t^\bullet$ either $s \in t^\bullet \setminus {}^\bullet t$ or $s \in {}^\bullet t$. Then (B)' from (N.b) and (N.c).

From (C.t) no notify_p^t message existed, thus (C.s)' and (C.t)'. As neither notify_p^u, lock_p^u, ackU_p^u nor ackL_p^u messages have been produced (C.e)'.

From (D.t) similarly (D.s)' and (D.t)'. As neither success^u_p, lock^u_p, nor unlock^u_p messages have been produced (D.e)'.

From (E.t) and (C.t) similarly (E.s)' and (E.t)'. (G.t) and (I.t) thus (G.s)' and (G.t)'. (G.e)' as neither lock^u_p nor ackU^u_p messages have been produced.

(H.t) thus (H.s)' and (H.t)'. (H.e)' as neither ackU^u_p nor ackL^u_p messages have been produced.

(I.t) thus (I.s)' and (I.t)'. (J.t), (G.t) and (I.t) thus (J.s)' and (J.t)'. (K.t) thus (K.s)' and (K.t)'.

(F.t) thus (F.s)' and (F.t)'. Assume $\text{token}^u_p \mathrel{\tilde{\in}} q$. For $u = t$ (F.t) is a contradiction with the performed step, thus $u \neq t$. For $p \in t^\bullet$ there is a contradiction with (F.t) and (N.d3). Thus (F.e)'.

(L.s)' and (L.e)' from (N.b) and (N.c). (M.*)' as all three arguments of the new state are empty.

Terms only improved for (N.a)', (N.d1)', (N.d2)', (N.d3)', (O.b)', (O.c)', (P.*)', (Q.a)' and (R.a)'.

Now consider (N.b)' and (N.c)', which are problematic as new newToken^t_s messages have been produced. Take any $s \in t^\bullet$. From (N.d3) there exists no transition $u \neq t$ with $s \in {}^\bullet u$ and $\text{firing}_u(\ldots) \mathrel{\tilde{\in}} q$. Thus (N.b)'. From (N.d1) there exists no transition $u \neq t$ for which $\text{firing}_u(\ldots) \mathrel{\tilde{\in}} q$ and $s \in u^\bullet$. Thus (N.c)'.

From (N.b) follows that no preplace p of t can be in a state $\text{waiting}_p(t, L, W)$ for any L and W. Thus (O.a)'.

Case $\text{empty}_s \xrightarrow{\{\text{newToken}^t_s\};\emptyset}_{A_s} \text{prenotify}_s$, $t \in {}^\bullet s$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' as no messages are produced. Thus also (*.e)'.

From (C.s) follows that no notify^u_s messages could have existed in q. Thus (C.*)'. Similarly from (D.s) follows (D.*)'. From (E.s) follows (E.*)'.

From (F.e) follows that no token^u_s message existed. Hence (F.*)'.

From (G.s) follows that no lock^u_s message existed. Thus (G.*)'. From (H.s) similarly (H.*)'. (I.s) thus (I.*)'. (J.s) thus (J.*)'. (K.s) thus (K.*)'.

(L.s)' as the only place which changed state was s and no second newToken^u_s existed, neither for $u = t$ as per (B) nor for $u \neq t$ per (L.e).

Terms only improved for (M.*)', (N.a)', (N.d1)', (N.d2)', (N.d3)', (O.*)', (P.*)', and (R.a)'. (N.b)' and (N.c)' as for the only place which changed state there existed a newToken^t_s message.

Take a posttransition u of s. If u is in a state $\text{firing}_u(U)$ then $s \in U$ would lead to a contradiction with (N.b). Thus $s \notin U$ and with $s \in {}^\bullet u$ then $s \in {}^\bullet u \setminus U$. Then from (N.a) follows that either a go_s^u or a token_s^u message exists. That leads to a contradiction via (K.e) and (F.e) respectively. If u is in a state $\text{locking}_u(L, l, T)$ then $s \in T$ leads to a contradiction with (M.a). The only remaining possibility is that u is in a state $\text{locking}_u(L, l, T)$ with $s \notin T$. Thus (Q.a)'.

Case $\text{prenotify}_s \xrightarrow{\{\text{internalNotify}^s\};\{\text{notify}_s^t \;|\; t \in s^\bullet\}}_{A_s} \text{unlocked}_s$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. From (C.s) follows that no notify_s^t messages existed yet, so (B)'.

(C.s)' trivially from the performed step. (C.t)' from (Q.a). (C.e)' from (G.s), (H.s), and (I.s) which respectively ensure that no lock_s^t, no ackU_s^t, and no ackL_s^t messages exist.

From (D.s) follows that no success_s^t message exists, thus (D.*)'. Similarly from (E.s) follows (E.*)'. From (F.s) follows (F.*)'. (G.s) thus (G.*)'. (H.s) thus (H.*)'. (I.s) thus (I.*)'. (J.s) thus (J.*)'. (K.s) thus (K.*)'. (L.s) thus (L.*)'.

Terms only improved for (M.*)', (N.*)', (O.*)', (P.*)', and (Q.a)'.

(R.a)' from the produced notify_s^t messages.

Case $\text{unlocked}_s \xrightarrow{\{\text{lock}_s^t\};\{\text{success}_s^t\}}_{A_s} \text{locked}_s(t, \emptyset)$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' as from (G.e) no success_s^t could have existed.

(C.s)' as the only transition u for which a notify_s^u message would be problematic is t. But per (G.e) no notify_s^t message exists. Thus also (C.e)'. (C.t)' as no state of a transition was changed.

From (D.s) no success_s^u message existed. Thus (D.*)'. From (E.s) similarly (E.*)'. (F.s) thus (F.*)'.

(G.s)' as the only transition u for which a lock_s^u message would be problematic is t. But the lock_s^t message was consumed and per (B) no second one exists. Thus also (G.e)'. (G.t)' as no state of a transition was changed and no ackL_p^u message was consumed.

From (H.s) no ackU_s^u message existed. Thus (H.*)'. (I.s) thus similarly (I.*)'. (J.s) thus (J.*)'. (K.s) thus (K.*)'. (L.s) thus (L.*)'.

Terms only improved for (M.a)', (M.c)', (N.*)', (O.*)', (Q.a)', and (R.a)'.

(M.b)' as the only problematic transitions could be t, but from (G.t) and (I.s) follows that t is in a state $\text{locking}_t(L, l, T)$ with $l = s$. (M.d)' as the consumed lock_s^t message has been replaced by the success_s^t message.

(P.a)' from the performed step. (P.b)' from (R.a). (P.c)' with the produced success_s^t message.

Case $\mathrm{locked}_s(t, L) \xrightarrow{\{\mathrm{lock}_s^u\};\emptyset}_{A_s} \mathrm{locked}_s(t, L \cup \{u\})$, $u \neq t, u \notin L$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' as no messages are produced. Thus also (*.e)'.

(C.s)' as the only transition added to L was u and from (G.e) no notify_s^u message existed. (C.t)' as no state of a transition was changed.

(D.s)' with the new value $L \cup \{u\}$. (D.t)' as no state of transition was changed.

From (E.s) follows that no loose_s^v message existed. Thus (E.*)'. Similarly (F.*)' follows from (F.s).

(G.s)' as the only transition v for which a lock_s^v message would be problematic is u. But the lock_s^u message was consumed and per (B) no second one exists. (G.t)' as no state of a transition was changed and no ackL_p^u was consumed.

From (H.s) follows that no ackU_s^v message existed. Thus (H.*)'. Similarly (I.*)' follows from (I.s).

(J.s)' with the new value $L \cup \{u\}$. Assume a $\mathrm{unlock}_s^v \mathrel{\widetilde{\in}} q$ exists. The only problematic case for (J.t)' is $v = u$ as no transition state was changed and only lock_s^u was consumed. However no unlock_s^u message exists as (J.s) and $t \neq u$ from the performed step lead to a contradiction otherwise. Thus (J.t)'.

(K.s)' with the new value $L \cup \{u\}$. (K.t)' as no state of a transition was changed.

From (L.s) follows that no $\mathrm{newToken}_s^v$ message existed. Thus (L.*)'.

Terms only improved for (M.a)', (M.c)', (N.*)', (O.*)', (P.b)', (P.c)', (Q.a)', and (R.a)'.

(M.b)' as the only value added to L was u and from (G.t) and (I.s) follows that u is in a state $\mathrm{locking}_u(L, l, T)$ with $l = s$. (M.d)' as the only consumed lock_s^v message has $v = u$ and u was added to L.

(P.a)' with the same argument as (M.b)'.

Case $\mathrm{locked}_s(t, L) \xrightarrow{\{\mathrm{unlock}_s^t\};\{\mathrm{success}_s^u\}}_{A_s} \mathrm{locked}_s(u, L \setminus \{u\})$, $u \in L$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' as per (D.s) no $\mathrm{success}_s^u$ message could have existed before.

(C.s)' as something was removed from L. (C.t)' as no state of a transition was changed. (C.e)' as no lock_s^v, no ackU_s^v, and no ackL_s^v messages have been produced.

(D.s)' trivially from the performed step. (D.t)' from (P.a). No unlock_s^u message could have existed as (J.s). Thus (D.e)'.

From (E.s) follows that no loose_s^v message can exist. Thus (E.*)'. Similarly from (F.s) follows (F.*)'.

Assume some lock_s^v message exists in q. For $v \neq t$ and $v \neq u$ nothing relevant changed in (G.s)'. For $v = t$ the unlock_s^t message was removed, but $t \notin L$ from Definition 5.2.2

so (G.s)' as far as a possible lock_s^t is concerned. For $v = u$ no lock_s^u message could have existed as (G.s) and $u \in L$. Thus (G.s)'.

(G.t)' as no state of a transition was changed and no ackL_p^u was consumed. (G.e)' as no ackU_p^v message was created.

From (H.s) follows that no ackU_s^v message existed. Thus (H.*)'. The same argument with (I.s) shows (I.*)'.

(J.s)' as the only problematic message could be unlock_s^t but it was consumed and per (B) no second one exists. (J.t)' as no state of a transition was changed and no lock_p^v message was consumed.

(K.s)' as the only problematic message could be go_s^t but such a message does not exists as per (J.e). (K.t)' as no state of a transition was changed.

From (L.s) follows that no $\mathrm{newToken}_s^v$ message existed. Thus (L.*)'.

Terms only improved for (M.a)', (N.*)', (O.*)', (P.a), (Q.a)', and (R.a)'.

To show (M.b)', assume some transition v exists such that $\mathrm{locking}_v(L', l, T) \mathrel{\tilde{\in}} q$ and $s \in T \setminus (L' \cup \{l\})$. If $v \neq t$ and $v \neq u$ then nothing relevant changed in (M.b)'. For $v = u$ there is a contradiction with (M.b) as $u \in L$. For $v = t$ (M.b)' holds as $t \notin L$. Thus (M.b)'.

The only transition problematic for (M.c)' is t, but from (J.t) either t is in a state $\mathrm{locking}_t(L, l, T)$ with $s \notin L$ or $\mathrm{lock}_s^t \mathrel{\tilde{\in}} q$ from which via (G.t) follows $\mathrm{locking}_t(L, s, T) \mathrel{\tilde{\in}} q$ where also $s \notin L$ per Lemma 5.2.1 or there must be an ackL_s^t message which is not possible as per (I.s). Thus (M.c)'.

(M.d)' as the removal of u from L is unproblematic with the newly produced $\mathrm{success}_s^u$ message.

(P.b)' from (P.a) as the only problematic transition is u which was in L earlier. (P.c)' as the unlock_s^t message was consumed but the first component of the state changed to u for which (P.c)' holds with the newly produced $\mathrm{success}_s^u$ message.

Case $\mathrm{locked}_s(t, \emptyset) \xrightarrow{\{\mathrm{unlock}_s^t\};\emptyset}_{A_s} \mathrm{unlocked}_s$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' as no messages are produced. Thus also (*.e)'.

(C.s)' from the performed step. (C.t)' as no state of a transition was changed.

(D.s)' as no $\mathrm{success}_s^t$ message existed per (J.e) and no other $\mathrm{success}_s^u$ message existed per (D.s). (D.t)' as no state of transition was changed.

From (E.s) follows that no loose_s^v message can exist. Thus (E.*)'. Similarly from (F.s) follows (F.*)'.

(G.s)' from the performed step. (G.t)' as no state of a transition was changed and no ackL_p^u was consumed.

From (H.s) follows that no ackU_s^v message existed. Thus (H.*)'. Using (I.s) follows (I.*)' similarly.

(J.s)' as the only possible unlock_s^u message has $u = t$. That message was consumed however, and per (B) no second one existed. (J.t)' as no state of a transition was changed and no lock_p^v was consumed.

(K.s)' as the only possible go_s^u message has $u = t$. From (J.e) however, no such message existed. (K.t)' as no state of a transition was changed.

From (L.s) follows that no $\mathrm{newToken}_s^u$ message existed. Thus (L.*)'.

Terms only improved for (M.a)', (M.b)', (M.d)', (N.*)', (O.*)', (P.a)', (P.b)' and (Q.a)'.

The only transition problematic for (M.c)' is t, but from (J.t) either t is in a state $\mathrm{locking}_t(L, l, T)$ with $s \notin L$ or $\mathrm{lock}_s^t \mathrel{\tilde{\in}} q$ from which via (G.t) follows $\mathrm{locking}_t(L, s, T) \mathrel{\tilde{\in}} q$ where also $s \notin L$ per Lemma 5.2.1 or there must be an ackL_s^t message which is not possible as per (I.s). Thus (M.c)'.

(P.c)' as the only unlock_p^u message consumed has $p = s$ and $u = t$ and the new state of s is unproblematic. (R.a)' from (P.b) as (J.e) excludes a go_s^t message.

Case $\mathrm{locked}_s(t, L) \xrightarrow{\{\mathrm{go}_s^t\};\{\mathrm{loose}_s^u \ \mid\ u\in s^\bullet, u\neq t\}}_{A_s} \mathrm{waiting}_s(t, L, s^\bullet \setminus \{t\})$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change since $\mathrm{go}_s^t \mathrel{\tilde{\in}} q$ implies via (K.t) that $\mathrm{firing}_t(T) \mathrel{\tilde{\in}} q$ for some T. (B)' as (E.s) ensured that no loose_s^v message existed before.

To show (C.s)' assume that some notify_s^v message existed. Then from (C.s) follows that $v \neq t$ and $v \notin L$. Thus $v \in s^\bullet \setminus \{t\}$ and $\mathrm{waiting}_s(t, L, s^\bullet \setminus \{t\})$ makes (C.s)' true for that message. Thus (C.s)'. (C.t)' as no state of a transition was changed. (C.e)' as no lock_s^v, no ackU_s^v, and no ackL_s^v messages have been produced.

(D.s)' as every message $\mathrm{success}_s^v$ must have $v = t$ per (D.s) and $\mathrm{success}_s^t$ is excluded by (K.e). (D.t)' as no state of a transition was changed. (D.e)' as no lock_s^v and no unlock_s^v messages have been produced.

No loose_s^v message could have existed in q as per (E.s). For the newly created messages (E.s)' follows from the performed step. (E.t)' follows from (P.a), (P.b) and (M.c) together with the observation that every $\mathrm{go}_s^v \mathrel{\tilde{\in}} q$ must have $v = t$ per (K.s).

From (F.s) follows that no token_s^v message can exist. Thus (F.*)'.

Assume some lock_s^v message existed in q. For $v \neq t$ the state $\mathrm{waiting}_s(t, L, s^\bullet \setminus \{t\})$ makes (G.s)' true for that message. For $v = t$ an unlock_s^t message would need to exist, which is not the case as per (K.e). Thus (G.s)'. (G.t)' as no state of a transition was changed and no ackL_p^v has been consumed. (G.e)' as no ackU_p^v message was created.

From (H.s) follows that no ackU_s^v message existed. Thus (H.*)'. Similarly (I.*)' follows from (I.s).

From (J.s) follows that every message unlock^v_s has $v = t$. But $\text{unlock}^t_s \mathrel{\widetilde{\notin}} q$ from (K.e). Thus (J.s)'. (J.t)' as no state of a transition was changed and no lock^v_p was consumed.

From (K.s) follows that every message go^v_s has $v = t$. But go^t_s was consumed and no second one existed as per (B). Thus (K.*)'.

From (L.s) follows that no newToken^u_s message existed. Thus (L.*)'.

Terms only improved for (M.a)', (N.b)', (N.c)', (N.d1)', (N.d2)', (N.d3)', (P.a)', (P.b)', (Q.a)', and (R.a)'.

For (M.b)' assume some transition v with $\text{locking}_v(L', l, T) \mathrel{\widetilde{\in}} q$ and $s \in T \setminus (L \cup \{l\})$ exists. If $v \neq t$ then $v \in s^\bullet \setminus \{t\}$ and (M.b)' holds for v. If $v = t$ then there would need to be an unlock^t_s message which is a contradiction to (K.e). Thus (M.b)'.

(M.c)' as the only problematic transition could be t which however is in state $\text{firing}_t(T)$ for some T as per (K.t). (M.d)' with the newly produced loose^u_s messages.

(N.a)' as the only go^v_p message consumed has $p = s$ and $v = t$ and s switched its state into $\text{waiting}_s(t, L, s^\bullet \setminus \{t\})$.

(O.a)' from (K.t). (O.b)' as $s^\bullet \setminus ((s^\bullet \setminus \{t\}) \cup \{t\}) = \emptyset$. (O.c)' with the newly produced loose^u_s messages.

(P.c)' as the only go^v_p message consumed has $p = s$ but the new state of s is unproblematic.

Case $\text{waiting}_s(t, L, W) \xrightarrow{\{\text{lock}^u_s\};\emptyset}_{A_s} \text{waiting}_s(t, L \cup \{u\}, W)$, $u \notin L, u \in W$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' as no messages are produced. Thus also (*.e)'.

(C.s)' as the only element added to L is u for which no notify^u_s message exists as per (G.e). (C.t)' as no state of a transition was changed.

From (D.s) no success^v_s message existed. Thus (D.*)'.

(E.s)' as only L was changed. (E.t)' as no notify^u_p messages were consumed and no state of a transition was changed.

From (F.s) follows that no token^v_s message can exist. Thus (F.*)'.

(G.s)' as the only element added to L is u, one lock^u_s message was consumed, no second one exists as per (B), and no unlock^v_p message was consumed. (G.t)' as no state of a transition was changed and no ackL^v_p was consumed.

(H.s)' as the only element added to L is u for which no ackU^u_s message exists as per (G.e). (H.t)' as no state of a transition was changed.

(I.s)' as the fact that u was added to L makes up for the consumed lock^u_s message. (I.t)' as no state of a transition was changed.

From (J.s) follows that no message unlock^v_s exists. Thus (J.*)'. Similarly from (K.s) follows (K.*)'. From (L.s) follows (L.*)'.

Terms only improved for (M.a)', (M.c)', (N.*)', (O.*)', (P.*)', (Q.a)', and (R.a)'.

(M.b)' as the only element added to L is u for which (G.t) and (I.t) guarantee that $\mathrm{locking}_u(L', l, T) \mathrel{\tilde{\in}} q$ such that $s \notin T \setminus (L' \cup \{l\})$. Regarding (M.d)', from (O.c) and $u \in W$ follows that a loose_s^u, an ackU_s^u, or an ackL_s^u message exists. If $\mathrm{loose}_s^u \mathrel{\tilde{\in}} q$ (M.d)', (G.e) excludes the ackU_s^u message, and if an ackL_s^u message exists, (I.t) guarantees that u is in an unproblematic state $\mathrm{locking}_u(L', l, T)$ for (M.d)' as $s \notin T$ and thus via Lemma 5.2.1 $l \neq s$. Thus (M.d)'.

Case $\mathrm{waiting}_s(t, L, W) \xrightarrow{\{\mathrm{ackL}_s^u\};\emptyset}_{A_s} \mathrm{waiting}_s(t, L \setminus \{u\}, W \setminus \{u\})$, $u \in L$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' as no messages are produced. Thus also (*.e)'.

(C.s)' as the only element removed from W is u for which (I.e) guarantees that no notify_s^u message exists. (C.t)' as no state of a transition was changed.

From (D.s) no $\mathrm{success}_s^v$ message existed. Thus (D.*)'.

(E.s)' as the only element removed from W is u for which (I.e) guarantees that no loose_s^u message exists. (E.t)' as no notify_p^u messages were consumed and no state of a transition was changed.

From (F.s) follows that no token_s^v message can exist. Thus (F.*)'.

(G.s)' as the only element removed from W is u which was in L before and for which per (G.s) no lock_s^u message exists. (G.t)' as no state of a transition was changed, the only consumed ackL_p^v message has $p = s$ and $v = u$, and no lock_s^u message exists per (G.s).

(H.s)' as the only element removed from W is u for which (I.e) guarantees that no ackU_s^u message exists. (H.t)' as no state of a transition was changed.

(I.s)' as for both W and L the only element removed is u for which one ackL_s^u message was consumed and no second one exists as per (B). (I.t)' as no state of a transition was changed.

From (J.s) follows that no message unlock_s^v exists. Thus (J.*)'. Similarly from (K.s) follows (K.*)'. From (L.s) follows (L.*)'.

Terms only improved for (M.a)', (M.c)', (M.d)', (N.*)', (O.a)', (P.*)', (Q.a)', and (R.a)'.

To show (M.b)', assume some transition v exists such that $\mathrm{locking}_v(L', l, T) \mathrel{\tilde{\in}} q$ and $s \in T \setminus (L' \cup \{l\})$. If $v = u$ then from (I.t) follows that $s \notin T$ and (M.b)' holds. For $v \neq u$ nothing relevant changed as only u was removed from W. Thus (M.b)'.

(O.b)' as the only element new to $s^\bullet \setminus (W \cup \{t\})$ is u for which (I.t) guarantees that $\mathrm{locking}_u(L', l, T) \mathrel{\tilde{\in}} q$ with $s \notin T$. (O.c)' as the only consumed message was ackL_s^u and u was removed from W.

Case $\mathrm{waiting}_s(t, L, W) \xrightarrow{\{\mathrm{ackU}_s^u\};\emptyset}_{A_s} \mathrm{waiting}_s(t, L, W \setminus \{u\})$, $u \notin L, u \in W$:

Then $\alpha(q')$ as follows: (A.b)', (B)', (*.e)', (C.*)', (D.*)', (E.*)', (F.*)', (H.t)', (I.t)', (J.*)', (K.*)', (L.*)', (M.*)', (N.*)', (O.*)', (P.*)', (Q.a)', and (R.a)' as in the previous case using (H.*) instead of (I.*) and the different message name, leaving (G.s)', (G.t)', (H.s)', and (I.s)' to be proven here.

(G.s)' as the only element removed from W is u for which (H.e) guarantees that no lock_s^u message exists. (G.t)' as no state of a transition was changed and no ackL_p^u was consumed in the step.

(H.s)' as the only element removed from W is u for which one ackU_s^u message was consumed and no second one exists per (B).

(I.s)' as the only element removed from W is u for which (H.e) guarantees that no ackL_s^u message exists.

Case $\mathrm{waiting}_s(t, \emptyset, \emptyset) \xrightarrow{\{\mathrm{internalPassToken}_s^t\};\{\mathrm{token}_s^t\}}_{A_s} \mathrm{empty}_s$:

Then $\alpha(q')$ as follows: (A.b)' as $\mathfrak{f}'$ didn't change. (B)' as per (F.s) no token_s^t message existed before.

From (C.s) follows that no notify_s^u messages existed. Thus (C.*)'. Similarly from (D.s) follows (D.*)'. From (E.s) follows (E.*)'.

From (F.s) follows that no token_s^u message existed before. For the new token_s^t message (F.s)' follows from the performed step. Thus (F.s)'. From (O.a) follows that $\mathrm{firing}_t(T) \mathrel{\tilde{\in}} q$ with $s \notin T$. Thus (F.t)'. From (L.s) follows that no $\mathrm{newToken}_s^u$ message exists. Thus (F.e)'.

From (G.s) follows that no lock_s^u message existed before. Thus (G.*)'. Similarly from (H.s) follows (H.*)'. From (I.s) follows (I.*)'. From (J.s) follows (J.*)'. From (K.s) follows (K.*)'. From (L.s) follows (L.*)'.

From (O.a) follows that t is in a state $\mathrm{firing}_t(T)$ with $s \notin T$. From (O.b) follows that all $u \in s^\bullet \setminus \{t\}$ are in a state $\mathrm{locking}_u(L', l, T)$ with $s \notin T$. Thus (M.a)' and (M.b)'.

Terms only improved for (M.c)', (M.d)', (N.b)', (N.c)', (N.d1)', (N.d2)', (N.d3)', (O.*)', (P.*)', (Q.a)', and (R.a)'.

(N.a)' with the newly produced token_s^t message. □

After having shown that every step of the implementation implies an equivalent step of the original net as well, the other direction is now shown: Every step of the net is also possible in the implementation. However this does not hold for all implementation states, but only for "normalised" implementation states, those which could be an initial implementation state as given in Definition 5.2.1 or Definition 5.2.2.

Definition 5.2.6

Let N be a plain net and let A_N be the FSM based implementation of it.

Let $n = |T^N| + |S^N|$.

The function $\mathfrak{F} : \mathcal{P}(S^N) \to Q^{A_N}$ is defined such that

$$\begin{aligned} \forall 1 \le i \le n.\ ((\pi_i(\mathfrak{F}(M)) = \text{locking}_t(\emptyset, \bot, \emptyset) \wedge t \in T^N) \vee \\ (\pi_i(\mathfrak{F}(M)) = \text{empty}_s \wedge s \in S^N \setminus M) \vee \\ (\pi_i(\mathfrak{F}(M)) = \text{prenotify}_s \wedge s \in M)) \\ \wedge\ \pi_{n+1}(\mathfrak{F}(M)) = \emptyset\ . \end{aligned}$$

The function $\mathfrak{F}$ is well defined as the result must lie within Q^{A_N} and is thus unique. Also applying $\mathfrak{f}$ after $\mathfrak{F}$ results in the identity.

Lemma 5.2.2

$\mathfrak{f}(\mathfrak{F}(M)) = M$.

Proof

Let $M \subseteq \mathcal{P}(S^N)$.

Take any $s \in M$. As $\mathfrak{F}$ maps into Q^{A_N}, there must, according to Definition 3.2.2, be some index i such that $\pi_i(\mathfrak{F}(M)) \in Q^{A_s}$. As $s \notin T^N$ and $s \in M$, that element must have $\pi_i(\mathfrak{F}(M)) = \text{prenotify}_s$. Take any $s \notin M$. Again there exists some i with $\pi_i(\mathfrak{F}(M)) \in Q^{A_s}$. And from $s \notin M$ then follows that $\pi_i(\mathfrak{F}(M)) = \text{empty}_s$. Similarly for every $t \in T^N$ follows that an i exists for which $\pi_i(\mathfrak{F}(M)) = \text{locking}_t(\emptyset, \bot, \emptyset)$. As $\mathfrak{F}(M)$ has distinct values at all these indices, the indices must be distinct, as $n = |S^N| + |T^N|$ the first n indices of $\mathfrak{F}(M)$ are uniquely determined. Also $\pi_{n+1}(\mathfrak{F}(M)) = \emptyset$.

Thus for all $s \in M$ follows that $\text{prenotify}_s \mathrel{\tilde{\in}} \mathfrak{F}(M)$ and as no messages exists $s \in \mathfrak{f}(\mathfrak{F}(M))$. For all $s \notin M$ follows that $\text{empty}_s \mathrel{\tilde{\in}} \mathfrak{F}(M)$ and as no transition is in $\text{firing}_t(T)$ for any T, also $s \notin \mathfrak{f}(\mathfrak{F}(M))$. □

Proposition 5.2.2

Let N be a plain net and let A_N be the FSM based implementation of it.

(i) $\mathfrak{F}(M_0^N) = q_0^{A_N}$ and

(ii) If $M\ [G\rangle_N\ M'$, then there exists a sequence $q_0, q_1, \ldots, q_n$ of states, a sequence $I_1, I_2, \ldots, I_n$, and a sequence $O_1, O_2, \ldots, O_n$ such that $q_0 \xrightarrow{I_1;O_1}_{A_N} q_1 \xrightarrow{I_2;O_2}_{A_N} \cdots \xrightarrow{I_n;O_n}_{A_N} q_n$, $\mathfrak{F}(M) = q_0$, $\mathfrak{F}(M') = q_n$, and there exists a j, $1 \le j \le n$ such that $i \ne j \Rightarrow O_i = \emptyset$ and $O_j = \{\text{fire}^t \mid t \in G\}$.

Proof

The allegedly existing sequence can be described uniquely by giving the performed input and internal actions. To make the execution sequence unique, assume an arbitrary total order $\leq$ on transitions. The following uses the notation $\mathrm{num}_i(X)$ to denote the i-th element of a totally ordered set, in particular to select the i-th smallest transition according to the just defined $\leq$ and to select the i-th smallest place according to the global order of places used in the construction of the FSM based implementation.

There exist x_1, x_2, x_3, x_4, and x_5 such that the following sequence fulfils all conditions.

$$\begin{aligned}
I_1 &= \{\mathrm{internalNotify}^s \mid s \in {}^\bullet G\} \\
I_2 &= \left\{\mathrm{notify}_p^t \;\middle|\; t \in ({}^\bullet G)^\bullet, p = \mathrm{num}_1({}^\bullet t \cap {}^\bullet G)\right\} \\
I_3 &= \left\{\mathrm{notify}_p^t \;\middle|\; t \in ({}^\bullet G)^\bullet, p = \mathrm{num}_2({}^\bullet t \cap {}^\bullet G)\right\} \\
&\ldots\ldots \\
I_{a-1} &= \left\{\mathrm{notify}_p^t \;\middle|\; t \in ({}^\bullet G)^\bullet, p = \mathrm{num}_{x_1}({}^\bullet t \cap {}^\bullet G)\right\} \\
I_a &= \left\{\mathrm{internalLock}_s^t \;\middle|\; t \in G, s = \mathrm{num}_1({}^\bullet t)\right\} \\
I_{a+1} &= \left\{\mathrm{lock}_s^t \;\middle|\; t \in G, s = \mathrm{num}_1({}^\bullet t)\right\} \\
I_{a+2} &= \left\{\mathrm{success}_s^t \;\middle|\; t \in G, s = \mathrm{num}_1({}^\bullet t)\right\} \\
I_{a+3} &= \left\{\mathrm{internalLock}_s^t \;\middle|\; t \in G, s = \mathrm{num}_2({}^\bullet t)\right\} \\
I_{a+4} &= \left\{\mathrm{lock}_s^t \;\middle|\; t \in G, s = \mathrm{num}_2({}^\bullet t)\right\} \\
I_{a+5} &= \left\{\mathrm{success}_s^t \;\middle|\; t \in G, s = \mathrm{num}_2({}^\bullet t)\right\} \\
&\ldots\ldots \\
I_{b-3} &= \left\{\mathrm{internalLock}_s^t \;\middle|\; t \in G, s = \mathrm{num}_{x_2}({}^\bullet t)\right\} \\
I_{b-2} &= \left\{\mathrm{lock}_s^t \;\middle|\; t \in G, s = \mathrm{num}_{x_2}({}^\bullet t)\right\} \\
I_{b-1} &= \left\{\mathrm{success}_s^t \;\middle|\; t \in G, s = \mathrm{num}_{x_2}({}^\bullet t)\right\} \\
I_b &= \left\{\mathrm{internalFire}^t \;\middle|\; t \in G\right\} \\
I_{b+1} &= \left\{\mathrm{go}_s^t \;\middle|\; t \in G, s \in {}^\bullet t\right\} \\
I_{b+2} &= \left\{\mathrm{loose}_p^t \;\middle|\; t \in ({}^\bullet G)^\bullet \setminus G, p = \mathrm{num}_1({}^\bullet t \cap {}^\bullet G)\right\} \\
I_{b+3} &= \left\{\mathrm{loose}_p^t \;\middle|\; t \in ({}^\bullet G)^\bullet \setminus G, p = \mathrm{num}_2({}^\bullet t \cap {}^\bullet G)\right\} \\
&\ldots\ldots \\
I_{c-1} &= \left\{\mathrm{loose}_p^t \;\middle|\; t \in ({}^\bullet G)^\bullet \setminus G, p = \mathrm{num}_{x_3}({}^\bullet t \cap {}^\bullet G)\right\} \\
I_c &= \left\{\mathrm{ackU}_p^t \;\middle|\; p \in {}^\bullet G, t \in \mathrm{num}_1(p^\bullet \setminus G)\right\} \\
I_{c+1} &= \left\{\mathrm{ackU}_p^t \;\middle|\; p \in {}^\bullet G, t \in \mathrm{num}_2(p^\bullet \setminus G)\right\} \\
&\ldots\ldots \\
I_{d-1} &= \left\{\mathrm{ackU}_p^t \;\middle|\; p \in {}^\bullet G, t \in \mathrm{num}_{x_4}(p^\bullet \setminus G)\right\}
\end{aligned}$$

$$\begin{aligned}
I_d &= \left\{ \text{internalPassToken}_s^t \;\middle|\; t \in G, s \in {}^\bullet t \right\} \\
I_{d+1} &= \left\{ \text{token}_s^t \;\middle|\; t \in G, s = \text{num}_1({}^\bullet t) \right\} \\
I_{d+2} &= \left\{ \text{token}_s^t \;\middle|\; t \in G, s = \text{num}_2({}^\bullet t) \right\} \\
&\cdots\cdots \\
I_{e-1} &= \left\{ \text{token}_s^t \;\middle|\; t \in G, s = \text{num}_{x_5}({}^\bullet t) \right\} \\
I_e &= \left\{ \text{internalDone}^t \;\middle|\; t \in G \right\} \\
I_{e+1} &= \left\{ \text{newToken}_s^t \;\middle|\; s \in G^\bullet \right\}
\end{aligned}$$

Finally, $j = b$. □

There are two additional properties of the implementation that will be necessary to prove correctness in Theorem 5.2.1. The first property is concerned with deadlocks, i.e. states where no further activity is possible, which the implementation should not introduce. The implementation must only deadlock in states which are related to states where a deadlock was present in the original net. The second property does a similar thing for livelocks, i.e. infinite sequences of unobservable activity. As the original net will be a plain net though, the original net cannot contain any livelocks, and hence the implementation should not include any either.

The implementation does not have a deadlock, if the original could have proceeded.

Proposition 5.2.3

Let N be a plain net and let A_N be the FSM based implementation of N. Let $q \in Q^{A_N}$ with $\alpha(q)$.

If there exists an A such that $\mathfrak{f}(q) \xrightarrow{A}_N$ then there also exist I, O and q' such that $q \xrightarrow{I;O}_{A_N} q'$. (Note that O does not need to have anything in common with A).

Proof

Assume no such O exists. Note first that also $O = \emptyset$ is perfectly acceptable, so no internal activity may occur either.

$\text{notify}_s^t \mathrel{\tilde{\in}} q$ would lead to some activity via (C.t). $\text{success}_s^t \mathrel{\tilde{\in}} q$ would lead to activity via (D.t). loose_s^t via (E.t) or (C.t). token_s^t via (F.t). lock_s^t via (G.s) or (J.s). ackU_s^t via (H.s). ackL_s^t via (I.s), (G.s) or (J.s). unlock_s^t via (J.s). go_s^t via (K.s). newToken_s^t via (L.s). Thus no message exists in q.

Also, there is no transition t is in a state of $\text{firing}_t(T)$ for any T. If $T = {}^\bullet t$ there is activity. Thus from (N.a) and the absence of messages there exists an $s \in {}^\bullet t \setminus T$ with $\text{waiting}_s(t, L, W) \mathrel{\tilde{\in}} q$. If $W = \emptyset$ there is activity. Thus from (O.c) some messages exist and there is a contradiction. Thus no transition t in a state $\text{firing}_t(T)$ can exist in q.

From $\mathfrak{f}(q) \xrightarrow{A}_N$ follows that there exists some G with $\mathfrak{f}(q)\ [G\rangle_N$. Now take $t \in G$. Clearly ${}^\bullet t \subseteq \mathfrak{f}(q)$. From Definition 5.2.4 then for every $s \in {}^\bullet t$ either $\text{prenotify}_s \mathrel{\tilde{\in}} q$, $\text{unlocked}_s \mathrel{\tilde{\in}} q$, $\text{locked}_s(u, L) \mathrel{\tilde{\in}} q$ for some u and L, or $\text{firing}_u(T) \mathrel{\tilde{\in}} q$ for some u and T. If $\text{prenotify}_s \mathrel{\tilde{\in}} q$ there is activity and $\text{firing}_u(T) \mathrel{\tilde{\in}} q$ is impossible as well. If $\text{unlocked}_s \mathrel{\tilde{\in}} q$ then from (R.a) and the absence of messages follows that $\forall u \in s^\bullet \exists L', l, T.\ \text{locking}_u(L', l, T) \mathrel{\tilde{\in}} q \wedge s \in T$. If $\text{locked}_s(u, L) \mathrel{\tilde{\in}} q$ then from (P.a), (P.b), and the absence of messages follows that $\forall u \in s^\bullet \exists L', l, T.\ \text{locking}_u(L', l, T) \mathrel{\tilde{\in}} q \wedge s \in T$. Repeating these arguments for each $s \in {}^\bullet t$ it follows that $\text{locking}_t(L', l, {}^\bullet t) \mathrel{\tilde{\in}} q$. If $l = \perp$ there is activity, thus $l \neq \perp$.

Then from (M.d) and the absence of messages follows that $\text{locked}_l(u, L') \mathrel{\tilde{\in}} q$ with $t \in L'$ and $u \neq t$. From (P.c) and the absence of messages then $\text{locking}_u(L'', l', T') \mathrel{\tilde{\in}} q$ with $l \in L''$. Assume $l' = \perp$ then together with $L'' \neq \emptyset$ follows from Lemma 5.2.1 that $T' = {}^\bullet u$ and there is activity. Thus $l' \neq \perp$ and from Lemma 5.2.1 $l < l'$.

Now consider a place $p \in {}^\bullet u$. Per (M.a) follows that either $\text{unlocked}_p \mathrel{\tilde{\in}} q$, $\text{locked}_p(\ldots) \mathrel{\tilde{\in}} q$, or $\text{waiting}_p(\ldots) \mathrel{\tilde{\in}} q$. With (O.a) however, the latter possibility is a contradiction with the fact that no $\text{firing}_v(\ldots) \mathrel{\tilde{\in}} q$.

From here on, the above arguments can be repeated, yielding a new l' each turn, and always strictly larger than the previous one. As N is finite however, at some point all places are exhausted. Thus there is a contradiction with the assumption that no activity is possible. □

The implementation does not have a livelock.

Proposition 5.2.4

Let N be a plain net and let A_N be the FSM based implementation of N. Let $q \in Q^{A_N}$ with $\alpha(q)$.

There exists no infinite sequence $I_1, I_2, \ldots$ such that $q \xrightarrow{I_1;\emptyset}_{A_N} \xrightarrow{I_2;\emptyset}_{A_N} \cdots$.

Proof

Assume an infinite sequence $I_1, I_2, \ldots$ such that $q \xrightarrow{I_1;\emptyset}_{A_N} \xrightarrow{I_2;\emptyset}_{A_N} \cdots$ exists.

As no visible output is allowed while the sequence is executing, no fire^t messages may be produced. The same step producing the fire^t messages however is the only step in which go_s^t messages are produced. Thus no step of the sequence may produce new go_s^t messages.

As N is finite and $\alpha(q)$ (B) holds, it follows that only finitely many go_s^t messages exist in q. As the sequence is assumed to be infinite however, there must be an I_i after which no further go_s^t messages are consumed.

The only step producing loose_s^u messages however consumes go_s^t messages. Again only finitely many loose_s^t messages exist, thus there must be some I_j after which no further loose_s^t messages are consumed. As all possibilities to produce an ackU_s^t or an ackL_s^t message require that a loose_s^t message is consumed, there is a point after which no further of these messages is produced and some I_k after which no ackL_s^t and no ackU_s^t is consumed.

Also the only step where a place enters its $\mathrm{waiting}_s(t, L, W)$ phase consumes a go_s^t message. Thus there must be some I_l after which no place enters its $\mathrm{waiting}_s(t, L, W)$ phase. Only finitely many places exist, and whenever a place enters its empty_s phase, it exited from a $\mathrm{waiting}_s(t, L, W)$ phase. Thus there must be some I_m after which no place enters its empty_s phase. As every place came from an empty_s state when it enters its $\mathrm{prenotify}_s$ phase, there must be a some I_n after which no place enters its $\mathrm{prenotify}_s$ phase. Thus there must be some I_o after which no place leaves its $\mathrm{prenotify}_s$ phase. As the creation of a notify_s^t message requires that s leaves its $\mathrm{prenotify}_s$ phase there must be some I_p after which no further notify_s^t messages are produced and some I_q after which no further notify_s^t messages are consumed.

After I_m as no place enters its empty_s phase, no further token_s^t messages are produced. Thus there is a I_r after which no further token_s^t messages are consumed. After that point no transition can enter its $\mathrm{firing}_t({}^\bullet t)$ state, as every transition must have at least one preplace (otherwise N would not be contact free), and the $\mathrm{firing}_t(T)$ phase starts with $T = \emptyset$. If no transition enters its $\mathrm{firing}_t({}^\bullet t)$ anymore there must be some I_s when the last transition leaves its $\mathrm{firing}_t({}^\bullet t)$ state and the last $\mathrm{newToken}_s^t$ message is produced. Thus there is some point I_t after which no further $\mathrm{newToken}_s^t$ message is consumed.

After I_j and I_q no loose_s^t and no notify_s^t messages are consumed, thus a transition in a state $\mathrm{locking}_t(L, l, T)$ can not change the T component any more. In particular no transition can enter a state $\mathrm{locking}_t(L, l, T)$ with $l \neq \perp$ and $T \neq {}^\bullet t$. Thus there is some I_u after which no transition leaves a $\mathrm{locking}_t(L, l, T)$ state with $l \neq \perp$ and $T \neq {}^\bullet t$. As leaving these states and consuming loose_s^t messages are the only two possibilities of producing unlock_p^t messages, there is some point after which no further unlock_p^t messages are produced and some I_v after which none are consumed any more.

As consuming unlock_s^t messages and leaving the $\mathrm{prenotify}_s$ state are the only possibilities for a place to enter its $\mathrm{unlocked}_s$ state and both are impossible after I_v and I_p, there must be a point after which no place enters its $\mathrm{unlocked}_s$ state any more. Thus there must also be some I_w after which no place leaves its $\mathrm{unlocked}_s$ state.

Consuming unlock_s^t messages and leaving the $\mathrm{unlocked}_s$ state of a place are the only possibilities for a $\mathrm{success}_s^t$ message to be produced. Both are impossible after I_v and I_w. Thus there must also be some I_x when no further $\mathrm{success}_s^t$ message is consumed.

As the only ways for a transition to enter a state of the form $\mathrm{locking}_t(L, \perp, {}^\bullet t)$ are consuming a notify_s^t message or consuming a $\mathrm{success}_s^t$ message, this does not happen after I_x and I_q. Thus there must be some point I_y after which no transition leaves a state of the form $\mathrm{locking}_t(L, \perp, {}^\bullet t)$. As lock_s^t messages are only produced when leaving such a state, no lock_s^t messages are produced after I_y and there is some I_z after which no lock_s^t message is consumed.

Thus no notify_s^t is consumed after I_q, no $\mathrm{success}_s^t$ is consumed after I_x, no loose_s^t is consumed after I_j, no token_s^t is consumed after I_r, no lock_s^t is consumed after I_z, no ackU_s^t is consumed after I_k, no ackL_s^t is consumed after I_k, no unlock_s^t is consumed after I_v, no go_s^t is consumed after I_i, no $\mathrm{newToken}_s^t$ is consumed after I_t. Thus there is a point after which no messages whatsoever are consumed.

Furthermore no $\mathrm{internalLock}^t$ can be performed after I_y, no $\mathrm{internalDone}^t$ can be performed after I_s, no $\mathrm{internalNotify}^s$ can be performed after I_o, no $\mathrm{internalPassToken}^s$ can be performed after I_m. Thus there is some point after which no step is possible anymore.

Therefore no infinite sequence $I_1, I_2, \ldots$ such that $q \xrightarrow{I_1;\emptyset}_{A_N} \xrightarrow{I_2;\emptyset}_{A_N} \cdots$ exists. □

Given the automaton based description of how to encode arbitrary nets into a distributed form, the following construction transforms those automatons back into nets, thereby finishing the distributed implementation transformation. The transformation back to nets proceeds in two separate steps, first the sequential FSMs representing the places and transitions of the original net are transformed into nets, then the parallel composition operator between state machines is replaced by a parallel composition operator between nets.

In the following construction, the power of multi-labelled transitions will be useful – for a short while – because there is no need to split up the parallel output of the automaton in an unnatural way. Later, all the net implementations of the generated FSMs will be combined again, and only singleton labelled transitions will remain. At that point, the resulting net is a plain τ-net.

Definition 5.2.7

Let A be a serial FSM.

The *net based implementation* of A is the net $N_A = (S^{N_A}, T^{N_A}, F^{N_A}, M_0^{N_A}, \ell^{N_A})$ with

- $S^{N_A} = \left\{\mathrm{state}_{A,q} \;\middle|\; q \in Q^A\right\} \cup \left\{\mathrm{input}_i \;\middle|\; i \in \Sigma_I^A\right\}$,
- $T^{N_A} = \left\{\mathrm{do}_{q,i,O,q'} \;\middle|\; q \xrightarrow{\{i\};O}_A q'\right\}$,
- $F^{N_A} = \left\{\begin{matrix}(\mathrm{state}_{A,q}, \mathrm{do}_{q,i,O,q'}), (\mathrm{input}_i, \mathrm{do}_{q,i,O,q'}),\\ (\mathrm{do}_{q,i,O,q'}, \mathrm{state}_{A,q'})\end{matrix} \;\middle|\; q \xrightarrow{\{i\};O}_A q', i \in \Sigma_I^A\right\} \cup$
  $\left\{(\mathrm{state}_{A,q}, \mathrm{do}_{q,i,O,q'}), (\mathrm{do}_{q,i,O,q'}, \mathrm{state}_{A,q'}) \;\middle|\; q \xrightarrow{\{i\};O}_A q', i \in \Sigma_\tau^A\right\}$,
- $M_0^{N_A} = \{\mathrm{state}_{A,q_0^A}\}$, and
- $\ell^{N_A}(\mathrm{do}_{q,i,O,q'}) = O$.

The set of *input places* of such a net is defined as $\mathfrak{I}(N) = \left\{\mathrm{input}_i \;\middle|\; i \in \Sigma_I^A\right\}$.

Also, the composition operator between state machines needs to be transformed into an operator between nets.

Definition 5.2.8

Let N and N' be two nets with clearly defined input places, i.e. nets produced by Definition 5.2.7 or by application of this definition.

Let $I = \mathfrak{I}(N) \cup \mathfrak{I}(N')$.

The *asynchronous parallel composition* of the two nets, $N\|N'$, is defined as the net $N\|N' = (S^{N\|N'}, T^{N\|N'}, F^{N\|N'}, M_0^{N\|N'}, \ell^{N\|N'})$ with

- $S^{N\|N'} = S^N \cup S^{N'}$,
- $T^{N\|N'} = T^N \cup T^{N'}$,
- $F^{N\|N'} = F^N \cup F^{N'} \cup \left\{(t, \mathrm{input}_o) \;\middle|\; t \in T^N \cup T^{N'}, \mathrm{input}_o \in I, o \in \ell^{N\|N'}(t)\right\}$,
- $M_0^{N\|N'} = M_0^N \cup M_0^{N'}$, and
- $\ell^{N\|N'}(t) = \begin{cases} \ell^N(t) \setminus \{i \mid \mathrm{input}_i \in I\} & \text{if } t \in T^N \\ \ell^{N'}(t) \setminus \{i \mid \mathrm{input}_i \in I\} & \text{if } t \in T^{N'} \end{cases}$.

The set of *input places* of the composition is defined as

$$\mathfrak{I}(N\|N') = I \setminus \left\{\mathrm{input}_i \;\middle|\; \exists t \in T^N.\ i \in \ell^N(t) \vee \exists t \in T^{N'}.\ i \in \ell^{N'}(t)\right\}$$

Using the above definition, the *net based implementation* of an asynchronous parallel composition of serial FSMs is defined as the asynchronous parallel composition of the net based implementations of the composed FSMs.

The net based implementation of a parallel composition of FSMs can be understood as a network of sequential machines in the sense of Definition 3.1.4 by adding the input_i places as buffer places also to the component which outputs to them.

The behavioural relation between the state machine composition and the net based implementation thereof is very close, as a bijective function between automaton states and reachable net states exists.

Definition 5.2.9

Let $A_1, A_2, \ldots, A_n$ be serial FSMs with pairwise matching action signatures, such that their asynchronous parallel composition $A_\|$ is 1-safe.

Let $N_1, N_2, \ldots, N_n$ be the respective net based implementations. Let $N_\|$ be the asynchronous parallel composition of the nets.

The function $\mathfrak{G} :\ Q^{A_\|} \to \mathcal{P}(S^{N_\|})$ is defined as

$$\mathfrak{G}(q) = \left\{\mathrm{state}_{A_i,\pi_i(q)} \;\middle|\; 1 \le i \le n\right\} \cup \{\mathrm{input}_o \mid o \in \pi_{n+1}(q)\}$$

For markings M where in each net $N_1, N_2, \ldots, N_n$ exactly one place of the form state_{A_i,q_i} is marked, the function $\mathfrak{g} :\ \mathcal{P}(S^{N_\|}) \to Q^{A_\|}$ is defined such that

$$\begin{aligned}(\forall 1 \le i \le n \exists q.\ \pi_i(\mathfrak{g}(M)) = q \wedge \mathrm{state}_{A_i,q} \in M)\\ \wedge\, \pi_{n+1}(\mathfrak{g}(M)) = \{o \mid \mathrm{input}_o \in M\}\end{aligned}$$

Lemma 5.2.3

Let $A_1, A_2, \ldots, A_n$ be serial FSMs with pairwise matching action signatures such that their asynchronous parallel composition $A_\|$ is 1-safe and such that $\Sigma_I^{A_\|} = \emptyset$.

Let $N_1, N_2, \ldots, N_n$ be the respective net based implementations. Let $N_\|$ be the asynchronous parallel composition of the nets.

Let M, M' be reachable markings of $N_\|$. Let q, q' be reachable states of $A_\|$.

(i) $\mathfrak{g}(\mathfrak{G}(q)) = q$
(ii) $\mathfrak{G}(\mathfrak{g}(M)) = M$
(iii) $\mathfrak{G}(q_0^{A_\|}) = M_0^{N_\|}$
(iv) $\mathfrak{g}(M_0^{N_\|}) = q_0^{A_\|}$
(v) $q \xrightarrow{I;O}_{A_\|} q' \Rightarrow \mathfrak{G}(q) \xrightarrow{O}_{N_\|} \mathfrak{G}(q') \vee (O = \emptyset \wedge \mathfrak{G}(q) \xrightarrow{\tau}_{N_\|} \mathfrak{G}(q'))$
(vi) $M \xrightarrow{O}_{N_\|} M' \Rightarrow \exists I.\ \mathfrak{g}(M) \xrightarrow{I;O}_{A_\|} \mathfrak{g}(M')$
(vii) $M \xrightarrow{\tau}_{N_\|} M' \Rightarrow \exists I.\ \mathfrak{g}(M) \xrightarrow{I;\emptyset}_{A_\|} \mathfrak{g}(M')$

Proof

(i): For each net N_i, $\text{state}_{A_i,\pi_i(q)} \in \mathfrak{G}(q)$ and $\forall x.\ \text{state}_{A_i,x} \in \mathfrak{G}(q) \Rightarrow x = \pi_i(q)$. Hence $\mathfrak{g}$ is defined for $\mathfrak{G}(q)$.

Also for $1 \leq i \leq n$, $\pi_i(\mathfrak{g}(\mathfrak{G}(q))) = \pi_i(q)$. Finally $\pi_{n+1}(\mathfrak{g}(\mathfrak{G}(q))) = \pi_{n+1}(q)$.

(ii): M is a reachable marking of $N_\|$. From Definition 5.2.7 follows that exactly one place of the from state_{A_i,q_i} is marked for every $1 \leq i \leq n$. Hence $\mathfrak{g}(M)$ is defined. In particular for every $1 \leq i \leq n$, $\pi_i(\mathfrak{g}(M)) = q_i$ and hence $\text{state}_{A_i,q_i} \in \mathfrak{G}(\mathfrak{g}(M))$. Finally $\text{input}_o \in M \Leftrightarrow \text{input}_o \in \mathfrak{G}(\mathfrak{g}(M))$.

(iii): Directly from Definition 5.2.7, Definition 5.2.8, and Definition 5.2.9.

(iv): From (iii) and (i).

(v): Consider first a singleton $I = \{a\}$.

Assume $q \xrightarrow{\{a\};O}_{A_\|} q'$. There is a unique automaton A_i with $a \in \Sigma_I^{A_i} \cup \Sigma_\tau^{A_i}$ where the action is either input or inner action.

If $a \in \Sigma_I^{A_i}$ then with $\Sigma_I^{A_\|} = \emptyset$ Definition 3.2.2 guarantees that $a \in \pi_{n+1}(q)$ and Definition 5.2.7 produced a transition $\text{do}_{\pi_i(q),a,O_a,\pi_i(q')}$ which consumes a token from input_i and one from $\text{state}_{A_i,\pi_i(q)}$. Hence this transition is enabled in the marking $\mathfrak{G}(q)$ as all these places are marked.

The transition produces a new token on $\text{state}_{A_i,\pi_i(q')}$ and, using Definition 5.2.8, one token on each place in $\left\{\text{input}_o \;\middle|\; o \in O_a \cap \Sigma_\tau^{A_\|}\right\}$. Only one place of the form $\text{state}_{A_i,x}$ is marked in $\mathfrak{G}(q)$. Thus the postplace of this form is either a preplace or empty. All postplaces of the form input_o must be empty as well, as otherwise the step would violate the assumption that $A_\|$ is 1-safe.

Furthermore, the label of $\mathrm{do}_{\pi_i(q),a,O_a,\pi_i(q')}$ which remains after all nets have been composed is $O_a \cap \Sigma_O^{A_\|}$, which, using Definition 3.2.2, equals O.

If $a \in \Sigma_\tau^{A_i}$ then Definition 5.2.7 produced a transition $\mathrm{do}_{\pi_i(q),a,O_a,\pi_i(q')}$ which has the single preplace $\mathrm{state}_{A_i,\pi_i(q)}$. Hence this transition is enabled in the marking $\mathfrak{G}(q)$.

The rest of the argument proceeds as above.

Now consider a non-singleton I. As the components have matching action signatures, no two components share input or output actions. Thus pre- and postplaces of all fired transitions are distinct and they can all fire in parallel.

(vi) and (vii):

As already noted above, in a reachable marking exactly one place of the form $\mathrm{state}_{A_i,x}$ will be marked in each net N_i. In particular this holds for M and M', thus $\mathfrak{g}$ is defined for both.

Instead of considering a whole step of $N_\|$ consider first a single transition firing.

Assume that $M\ [\{t\}\rangle_{N_\|}\ M'$. Let i be the index of the net where t originated.

If t has some preplace of the form input_a, then per Definition 5.2.7, $q_i \xrightarrow{\{a\};O_i}_{A_i} q_i'$ for some O_i (possibly empty). Also t will have one other preplace, namely state_{A_i,q_i}. Furthermore t will have the postplace $\mathrm{state}_{A_i,q_i'}$ and from Definition 5.2.8 also one postplace input_o for each $o \in O_i \cap \Sigma_\tau^{A_\|}$. Note that $O_i = \ell^{N_i}(t)$ and using Definition 5.2.8 $\ell^{N_\|}(t) = O_i \cap \Sigma_O^{A_\|}$, which is the O visible in the net step or the empty set in case of a τ-step.

As all preplaces of t are marked in M, all postplaces are marked in M', and $\Sigma_I^{A_\|} = \emptyset$ Definition 5.2.9 enforces that $\pi_i(\mathfrak{g}(M)) = q_i$, $a \in \pi_{n+1}(\mathfrak{g}(M))$, $\pi_i(\mathfrak{g}(M')) = q_i'$, and $\pi_{n+1}(\mathfrak{g}(M')) = \pi_{n+1}(\mathfrak{g}(M)) - \{a\} + O_i \cap \Sigma_\tau^{A_\|}$. Also $a \in \Sigma_I^{A_i}$ and $\ell^{N_\|}(t) = O$ and hence with all other components non-moving, the composition can perform $\mathfrak{g}(M) \xrightarrow{\{a\};O}_{A_\|} \mathfrak{g}(M')$.

If t has no preplace of the form input_a, then per Definition 5.2.7, $q_i \xrightarrow{\{a\};O_i}_{A_i} q_i'$ with $a \in \Sigma_\tau^{A_i}$ and $a \in \Sigma_\tau^{A_\|}$. The transition t will have exactly one preplace, namely state_{A_i,q_i}.

All considerations about postplaces and output are as above.

As all preplaces of t are marked in M and unmarked in M', Definition 5.2.9 enforces that $\pi_i(\mathfrak{g}(M)) = q_i$, $\pi_i(\mathfrak{g}(M')) = q_i'$. Hence with all other components non-moving, the composed automaton can perform $\mathfrak{g}(M) \xrightarrow{\{a\};O}_{A_\|} \mathfrak{g}(M')$.

If a set of transition G is firing, no two transitions share a common pre- or postplace as they are independent. Thus the respective state machine components consume different input messages and can proceed in parallel. □

One other fact is still missing, namely that the given implementations are indeed distributed. Every net based implementation as defined in this thesis is distributed.

Lemma 5.2.4

Let N be a net which has been produced by Definition 5.2.7 or by application of Definition 5.2.8.

N is distributed.

Proof

First case: N has been produced by Definition 5.2.7 from an automaton A.

Every transition always consumes one token from a place of the form $\text{state}_{A,q}$ and produces a token on one such place. Initially there is exactly one place of that form marked. Thus $M \in [M_0^N\rangle \Rightarrow |M \cap \left\{\text{state}_{A,q} \,\middle|\, q \in Q^A\right\}| = 1$. As every transition consumes one token from such a place, no two transitions can ever fire in parallel. Hence the trivial distribution locating all elements on the same location makes the net distributed.

Second case: N has been produced by Definition 5.2.8 and is actually of the form $N' \| N''$.

By induction over the application depth of Definition 5.2.8, it can be assumed that both N' and N'' are distributed by distributions $\mathscr{D}'$ and $\mathscr{D}''$ respectively.

Without loss of generality it can be assumed that $\mathscr{D}'$ and $\mathscr{D}''$ map to disjunct sets of locations. A valid distribution for $N' \| N''$ is then $\mathscr{D}' \cup \mathscr{D}''$ where functions have been understood as relations. To show that this is indeed a correct distribution, all transitions must be co-located with their preplaces and every pair of concurrently firing transitions must not be co-located.

Assume a transition t and its preplace p are not co-located. As the only entries in the flow-relation of $N' \| N''$ which were not present in N' or N'' go from transitions to places the preplace relation between t and p must have been present in N' or N'', which violates the assumption that the respective net is distributed.

Assume two transitions t and u fire in parallel. If they both belong to the same net, N' or N'', then that net is not distributed, violating the assumptions. If they belong to different nets they are not co-located as $\mathscr{D}'$ and $\mathscr{D}''$ map to disjunct sets of locations. □

Putting it all together, the main theorem can finally been proven.

Theorem 5.2.1

Let N be a plain net. Let N' be the net based implementation of the FSM based asynchronous implementation of N. Let N'' be the net N' where every label of the form $\{\text{fire}^t\}$ has been replaced by the label $\{t\}$.

Then N'' is distributed and completed step trace equivalent equivalent to N.

Proof

N' is distributed as per Lemma 5.2.4. As this property is independent of labelling, so is N''.

Let $A_\parallel$ be the FSM based asynchronous implementation of N.

"$\mathrm{CST}(N'') \subseteq \mathrm{CST}(N)$": Assume $a_1a_2a_3\ldots a_n \in \mathrm{CST}(N'')$ and $a_n \neq 0$ and $a_n \neq \delta$.

Then per definition $M_0^{N''} \xRightarrow{a_1a_2a_3\ldots a_n}_{N''} M$ for some M.

Undoing the renaming and applying Lemma 5.2.3 one obtains that $A_\parallel$ can perform a sequence of actions where the only visible outputs are of the form $\{\mathrm{fire}^t \mid t \in a_i\}$ in correct order and arrives at $\mathfrak{g}(M)$.

From Proposition 5.2.1 then follows that $M_0^N \xrightarrow{a_1}_N \xrightarrow{a_2}_N \xrightarrow{a_3}_N \cdots \xrightarrow{a_n}_N \mathfrak{f}(\mathfrak{g}(M))$ and thus $a_1a_2a_3\ldots a_n \in \mathrm{CST}(N)$.

Now assume that $a_1a_2a_3\ldots a_n0 \in \mathrm{CST}(N'')$. Then per definition $M_0^{N''} \xRightarrow{a_1a_2a_3\ldots a_n}_{N''} M$ for some M such that $M \not\xrightarrow{\tau}_{N''}$ and $M \not\xrightarrow{A}_{N''}$ for all A.

Using the reasoning above, $M_0^N \xrightarrow{a_1}_N \xrightarrow{a_2}_N \cdots \xrightarrow{a_n}_N \mathfrak{f}(\mathfrak{g}(M))$.

Assume that $\mathfrak{f}(\mathfrak{g}(M)) \xrightarrow{A}_N$. Then from Proposition 5.2.3 follows that $\mathfrak{g}(M) \xrightarrow{I;O}_{A_\parallel}$ for some I and O. If $O = \emptyset$ then Lemma 5.2.3 leads to $M \xrightarrow{\tau}_{N''}$, and if $O \neq \emptyset$ then Lemma 5.2.3 leads to $M \xrightarrow{A}_{N''}$ both of which violate the assumptions. Hence $\mathfrak{f}(\mathfrak{g}(M)) \not\xrightarrow{A}_N$ and as N is a plain net also $\mathfrak{f}(\mathfrak{g}(M)) \not\xrightarrow{\tau}_N$ and $a_1a_2a_3\ldots a_n0 \in \mathrm{CST}(N)$.

Now assume that $a_1a_2a_3\ldots a_n\delta \in \mathrm{CST}(N'')$. Then from Lemma 5.2.3 follows that $A_\parallel$ can reach a state where an infinite sequence of moves without output is possible, contradicting Proposition 5.2.4. Thus no such trace can exist in $\mathrm{CST}(N'')$.

"$\mathrm{CST}(N) \subseteq \mathrm{CST}(N'')$": Assume $a_1a_2a_3\ldots a_n \in \mathrm{CST}(N)$ and $a_n \neq 0$ and $a_n \neq \delta$.

Then per definition $M_0^N \xRightarrow{a_1a_2a_3\ldots a_n}_N M$ for some M.

Then via Proposition 5.2.2 $A_\parallel$ can perform a sequence of state transitions where the only visible outputs are of the form $\{\mathrm{fire}^t \mid t \in a_i\}$ in correct order and arrives in the state $\mathfrak{F}(M)$.

From Lemma 5.2.3 follows that N' can perform $M_0^{N'} \xRightarrow{\{\mathrm{fire}^t \mid t\in a_1\}\cdots\{\mathrm{fire}^t \mid t\in a_n\}}_{N'} \mathfrak{G}(\mathfrak{F}(M))$.

Via the renaming then $M_0^{N''} \xRightarrow{a_1\cdots a_n}_{N''} \mathfrak{G}(\mathfrak{F}(M))$ and $a_1a_2a_3\ldots a_n \in \mathrm{CST}(N'')$.

Now assume that $a_1a_2a_3\ldots a_n0 \in \mathrm{CST}(N)$. Then per definition $M_0^N \xRightarrow{a_1a_2a_3\ldots a_n}_N M$ for some M such that $M \not\xrightarrow{\tau}_N$ and $M \not\xrightarrow{A}_N$ for all A.

As above, $A_\parallel$ can reach $\mathfrak{F}(M)$ while producing the correct outputs. From Proposition 5.2.4 follows that if $A_\parallel$ continues from $\mathfrak{F}(M)$ by performing steps without output, it will ultimately reach a state q where it cannot perform any more silent moves. From Proposition 5.2.1 follows that $\mathfrak{f}(q) = \mathfrak{f}(\mathfrak{F}(M))$. Furthermore from Lemma 5.2.2 follows that $\mathfrak{f}(\mathfrak{F}(M)) = M$.

From Lemma 5.2.3 follows that N' can perform $M_0^{N'} \xRightarrow{\{\mathrm{fire}^t \mid t\in a_1\}\cdots\{\mathrm{fire}^t \mid t\in a_n\}}_{N'} \mathfrak{G}(q)$. Via the renaming then $M_0^{N''} \xRightarrow{a_1\cdots a_n}_{N''} \mathfrak{G}(q)$. And from the same Lemma 5.2.3 follows that N' and N'' cannot perform any silent moves from $\mathfrak{G}(q)$.

Now assume $\mathfrak{G}(q) \xrightarrow{A}_{N''}$ for some $A \neq \emptyset$. Then N' can proceed with $\{\text{fire}^t \mid t \in A\}$ and from Lemma 5.2.3 follows that $A_{\|}$ could proceed via $q \xrightarrow{I;\{\text{fire}^t \mid t \in A\}}_{A_{\|}}$ for some I. But then Proposition 5.2.1 shows that N could have proceeded via $\mathfrak{f}(q) \xrightarrow{A}_{N}$ and using $\mathfrak{f}(q) = M$ there is a contradiction to the assumption that it cannot. Thus $\mathfrak{G}(q) \stackrel{A}{\not\longrightarrow}_{N''}$.

Hence $a_1 a_2 a_3 \ldots a_n 0 \in \mathrm{CST}(N'')$.

Finally no trace ending in δ can exist in $\mathrm{CST}(N)$, as N is plain. $\square$

6 Conclusion

6.1 Discussion

This thesis has shown that all finite plain 1-safe Petri nets can be implemented in a distributed fashion while preserving behaviour up to step trace equivalence. This section discusses some possible interpretations of this result.

First note that of the three restrictions imposed upon the original net, only one is significant. 1-safety can be ensured by introducing co-places in a first step. Plainness can be introduced by relabelling all transitions. Undoing that relabelling after the implementation has been generated should produce a net equivalent to the non-plain original.

The restriction to finite nets however is a serious limitation, which can not be solved trivially due to various possibilities for livelock. The simplest case is just an infinite set of transitions of which each has a single preplace which is marked initially. Then the protocol given in Section 5 makes infinitely many internalNotify^s actions possible in sequence. This livelock is artificial however, as it only occurs due to voluntary interleaving of all these actions. But even if completed step trace equivalence could somehow be mended not to detect these kind of "parallel" livelocks, more serious cases exist, due the following problem.

The implementation is correct because step trace equivalence does allow the system to perform steps in sequence which were parallel in the original. This fact could be seen as a violation of the usual intuition. Usually, when including the interleavings of parallel actions into the permissible traces of a system, one assumes that such interleavings occur due to imperfection in timing. As the concept of "same point in time" is dubious in distributed systems anyway, this only seems natural. However, the implementation given in this thesis uses these interleavings in a different way. Actions which were independent before can occur in strict sequence in some runs of the implementation. This difference becomes apparent if one considers the causal structure of actions. Two actions which were parallel in the original system are never causally dependent upon each other. In the implementation such a dependency can arise spontaneously however.

Consider the net in Figure 4.5 and the step trace $\{v\}\{t\}$. In the original net, no token was passed from t to v or vice versa, the two transitions fired causally independent. In the implementation however, the following scenario can unfold. u sends a lock_p^u to p which subsequently grants the lock to u. Then v sends a lock_q^v to q which grants the lock to v. Then t attempts to lock p but receives no immediate answer as p is locked to u. Then u tries to lock q but also receives no immediate answer as q is locked to v. Then v fires, consuming the token on q, which in turn produces a loose^u_q message. This message then causes u to release its lock on p, which subsequently grants the lock to t which finally fires. This firing of t is causally dependent on the firing of v. Technically this can be shown by tracing the ancestry of the tokens finally consumed by t and showing that some of them stem from the tokens produced by v.

Some cosmetics can be applied by splitting the firing of transitions into an invisible part which handles the protocol with the preplaces and only then performing the visible output, thus making the firing of t again causally independent of the firing of v. However these cosmetics cannot solve the underlying problem that t is causally dependent upon the token initially placed on q. While this may seem harmless in the example, and poses no problem for finite Petri nets, consider an infinite chain of transitions as if Figure 4.5 had been repeated downwards. Then infinitely long causal chains can evolve, leading to a true sequential livelock while they unravel.

In practice however, infinite systems do not occur. Even long causal chains can only occur if a long chain of transitions in direct conflict (two transitions are both enabled and share a common preplace) existed in the original net. The garbling of the causal structure of the original system should not matter in practice either, as most environments will not care whether two actions have been performed in sequence due to imperfections in timing or due to true causality.

Also, if Petri net model a real system, it is often possible to substitute profound algorithms where the net employed non-determinism. The most interesting place for this transformation in the construction given in this thesis is the production of a success^u_s message after a place receives an unlock^t_s message. While all choices for u are correct as per Theorem 5.2.1, some algorithms might lead to better performance in practice. Possible options include preferring the longest waiting transition (suggested by [5]), the transition which already holds the most locks, or the transition which has the least remaining locks to acquire. The latter two options correspond to a static priority over all transitions, whereas the first option can be implemented by saving the set of waiting transitions in a queue of some sort.

On the theoretical side, this thesis has shown that arbitrary behaviours can be implemented distributedly under completed step trace equivalence and thus under all coarser equivalences as well. It is an interesting question which equivalence relations allow distributed implementations and where in the linear-time branching-time spectrum the boundary for distributed implementability lies. This thesis has removed a part of the grey area on the coarse side, limiting the position of the boundary to be not coarser than completed step trace equivalence and, with [7], not finer than step readiness equivalence. Also, the present thesis hints that causality can not be preserved in a distributed implementation, while parallelism can.

Additionally this thesis proposed a new model of asynchronous systems, which is closely related to a certain class of distributed Petri nets, but allows for a more compact representation of many distributed algorithms.

This thesis has also shown, to me at the very least, that the proof method employed here (and also in [8] and [7]) will be inadequate if the implementations of Petri nets include any more complexity. My motivation to employ the Isabelle/HOL tool was mainly fuelled by the anticipation of the proof of Proposition 5.2.1. Unfortunately it was not possible to verify that proof using Isabelle/HOL within the given time frame. Indeed I found using Isabelle/HOL is much more time consuming than I assumed initially due to two problems. First, the automated proof and term simplification methods within Isabelle/HOL take impractical amounts of time if the terms get large, as it is the case with the combination of all terms of the main invariant α. That problem will clearly be solved within a few years, if not by better algorithms, then by faster hardware. Second, due to the formality of formal tools, one feels pressed to proof trivialities (usually turning out not be trivial at all if considered in a strict formal setting), which distracts from the main line of proof.

Instead of hoping for better tool support in the near future, it might be possible to design protocols like the one in this thesis using a synchronous specification language, say CCS, and then refine it towards asynchrony stepwise, while also refining the invariants. I designed the construction directly in an asynchronous model however, so a synchronous version did not seem natural. Also I feel that designing algorithms directly in an asynchronous model will often lead to a higher grade of parallelism then a refinement of a synchronous algorithm usually yields. Using results like the one in this thesis however, it might at some point not be necessary any more to implement parallelism “by hand” at all. Instead well understood and performant protocols might be available for all practical problems.

6.2 Related Work

The question whether, and if how, it is possible to implement synchronous system descriptions in a distributed and asynchronous fashion has been asked and answered in a variety of ways before this thesis already.

In [13], Lynch has collected quite a lot of impossibility results about distributed systems, many of which concern asynchronous systems. In [7], van Glabbeek, Goltz and myself have answered the question negatively for the model of Petri nets, if branching-time is assumed, as already discussed in Section 4. In [12], Hopkins also identifies some synchronous behaviours which can not be implemented in a distributed fashion, again using Petri nets but employing a different notion of distributed.

The works [1], [2], [18], [16], and [10] by de Boer, Gorla, Klop, Nestmann, and Palamidessi compared asynchronous variations of the process algebras CCS and ACP and the π-calculus with each other and also with the original versions of the calculi. They then attempted to implement seemingly less asynchronous variants in more asynchronous ones. Depending on the used equivalence relation and the exact nature of the modifications applied to the process algebra, they reached both impossibility results and working implementations. These process algebra centric works have the advantage that their implementations can use the expressive power process algebras provide. On the other hand, the high level of abstraction sometimes hides synchronous features in the depths of the operator semantics, like the atomic choice happening when multiple receivers exist for a single message.

In [3], Fischer and Janssen identify systems which behave equivalently, up to failures semantics, whether they are implemented using synchronous or asynchronous communication, with the goal of using synchronous specifications to build asynchronous systems. In [21], Rabin and Lehmann give a randomised algorithm which solves the dining philosophers problem in an asynchronous and symmetric fashion. There are quite a lot of other results which solve one or the other real-world problem in a distributed and asynchronous fashion, many of which have been collected by Lynch in [14]. Indeed many methods employed in practice to build asynchronous systems are often neglected in theoretical literature which includes impossibility results, in particular the possibility of using a approximately correct local clock and thus timeouts and the possibility of using probabilistic choices.

Compared to models in the literature, asynchronously composed state machines as defined in this thesis are one of the most asynchronous models proposed. They are related most closely to the three following models.

In [22] W. Reisig introduced networks of sequential machines. While the differences have already been outlined in Section 3, I omitted a detail there to keep the implicit assumption that tokens do not carry any meaningful information implicit. In particular I dropped the free-choice condition on the grounds that otherwise a sequential component could not react differently on different input. When a Petri net models the control flow of a complicated system however, it is often the case that tokens do not just carry the information of their presence but additional data. In particular, where the Petri net only contains a non-deterministic and free choice, the real system might employ an algorithm which decides differently depending on the concrete information carried in the token. If a network of sequential machines as defined by Reisig behaves correctly, this correctness is independent of those hidden data and algorithms. The present thesis however needs an explicit representation of the data and the algorithms relevant to the implementation protocol to show its correctness.

Another model for asynchronous systems are the IO-Automata of Lynch and Tuttle [15]. They are however not asynchronous according to my intuition. While the sending of a message can only be controlled by a single component and the sender can not be blocked due to input enabledness of all receivers, the model ignores the possibility of message overtaking. The system sketched in Figure 6.1, if composed using IO-Automata semantics, cannot reach the error state, while it can do so if composed using asynchronous state machine composition. A similar problem also exists in the model used for example by Gouda, Chow and Lam in [11], which they call “communicating finite state machines”, as they couple sequential machines using FIFO-buffers, again making some forms of message overtaking impossible.

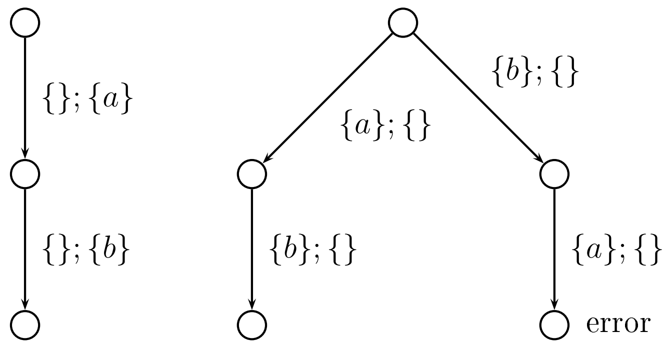

Figure 6.1: Two sequential components which, depending on the composition operator, sometimes reach the undesired state labelled "error"

Considering all results about asynchronous systems, the overall picture is far from clear. Apart from countless detailed ones the following large questions remain:

- How do the various models of asynchronous systems relate? Does asynchrony carry over into, for example, Petri net semantics of asynchronous process algebras.
- Which fundamental boundaries between the different shades of asynchrony exist and where exactly are they?
- Which models of asynchronous systems are relevant in practice?
- Which equivalence relations are best suited to describe the behaviours an asynchronous system or a component thereof can exhibit?
- How to transform the knowledge about asynchronous systems into practical tools like compilers or hardware synthesisers?
- How to build, verify and test large asynchronous systems?
- Which is the grand unifying theory answering all these questions?

Bibliography

[1] Frank S. de Boer and Catuscia Palamidessi. Embedding as a tool for language comparison: On the CSP hierarchy. In J.C.M. Baeten and J.F. Groote, editors, Proc. 2nd International Conference on *Concurrency Theory* (CONCUR'91), Amsterdam, The Netherlands, volume 527 of LNCS, pages 127–141. Springer, 1991. 68

[2] Frank S. de Boer, Catuscia Palamidessi, and Jan Willem Klop. Asynchronous communication in process algebra, 1992. Extended abstract. 68

[3] Clemens Fischer and Wil Janssen. Synchronous development of asynchronous systems. In Proc. 7th International Conference on *Concurrency Theory* (CONCUR'96), pages 735–750, London, UK, 1996. Springer. 69

[4] Rob J. van Glabbeek. The linear time – branching time spectrum I: The semantics of concrete, sequential processes. 3

[5] Rob J. van Glabbeek. Personal communication. 67

[6] Rob J. van Glabbeek. The linear time - branching time spectrum II. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93)*, pages 66–81, London, UK, 1993. Springer-Verlag. 3

[7] Rob J. van Glabbeek, Ursula Goltz, and Jens-Wolfhard Schicke. On synchronous and asynchronous interaction in distributed systems. Technical Report 2008-04, TU Braunschweig, 2008. Extended abstract in Proceedings 33rd *International Symposium on Mathematical Foundations of Computer Science* (MFCS 2008), Toruń, Poland, August 2008 (E. Ochmański & J. Tyszkiewicz, eds.), LNCS 5162, Springer, 2008, pp. 16-35. 4, 7, 21, 23, 67, 68

[8] Rob J. van Glabbeek, Ursula Goltz, and Jens-Wolfhard Schicke. Symmetric and asymmetric asynchronous interaction. Technical Report 2008-03, TU Braunschweig, 2008. Extended abstract in Proceedings 1st *Interaction and Concurrency Experience* (ICE'08) on *Synchronous and Asynchronous Interactions in Concurrent Distributed Systems*, to appear in *Electronic Notes in Theoretical Computer Science*, Elsevier. 13, 68

[9] Ursula Goltz. Personal communication.

[10] Daniele Gorla. On the relative expressive power of asynchronous communication primitives. In L. Aceto and A. Ingólfsdóttir, editors, *Proc. of 9th Intern. Conf. on Foundations of Software Science and Computation Structures (FoSSaCS '06)*, volume 3921 of LNCS, pages 47–62. Springer, 2006. 68

[11] Mohammed G. Gouda, C. H. Chow, and S. S. Lam. Livelock detection in networks of communicating finite state machines. Technical Report 84-10, University of Texas, Austin, Department of Computer Science, 1984. 69

[12] Richard P. Hopkins. Distributable nets. In *Advances in Petri Nets 1991*, volume 524 of LNCS, pages 161–187. Springer, 1991. 68

[13] Nancy A. Lynch. A hundred impossibility proofs for distributed computing. In *Proc. of the 8th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–28, New York, NY, 1989. ACM Press. 68

[14] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996. 69

[15] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1988. 69

[16] Uwe Nestmann. What is a 'good' encoding of guarded choice? *Information and Computation*, 156:287–319, 2000. 68

[17] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. 6

[18] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on *Principles of Programming Languages (POPL '97)*, pages 256–265. ACM Press, 1997. 68

[19] Carl Adam Petri. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. 4

[20] Vaughan R. Pratt. The pomset model of parallel processes: Unifying the temporal and the spatial. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 180–196, London, UK, 1985. Springer. 4

[21] Michael O. Rabin and Daniel J. Lehmann. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In Anthony W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, chapter 20, pages 333–352. 1994. An extended abstract appeared in *Proceedings of POPL'81*, pages 133–138. 69

[22] Wolfgang Reisig. Deterministic buffer synchronization of sequential processes. *Acta Informatica*, 18:115–134, 1982. 13, 14, 69

A Appendix

The following contains formal proofs in the Isabelle/HOL system for some of the constructions and lemmas used in this thesis.

theory *DrahflowTools*
imports *Main Multiset*
begin

Used for top-down proof development and as a filler for left-out parts.

axioms *proofHole*: P

lemma *eq-cong-fun-app*: $[\![x = y]\!] \Longrightarrow f\ x = f\ y$ **by** *simp*
lemma *directContradiction*: $[\![\neg\ P \Longrightarrow \mathit{False}]\!] \Longrightarrow P$ **by** *blast*

lemma *ballE-in*: $[\![\forall x{\in}A.\ Q\ x;\ x \in A;\ Q\ x \Longrightarrow P\ x]\!] \Longrightarrow P\ x$ **by** *blast*
lemma *ballE-in-double*: $[\![\forall x{\in}A.\ \forall y{\in}B.\ Q\ x\ y;\ x \in A;\ y \in B;\ Q\ x\ y \Longrightarrow P\ x\ y]\!] \Longrightarrow P\ x\ y$
by *blast*
lemma *bexToEx*: $[\![\exists\, x \in A.\ P\ x]\!] \Longrightarrow \exists\, x.\ P\ x$ **by** *blast*

lemma *some-connect*: $\bigwedge P\ Q.\ [\![\exists\, x.\ P\ x;\ \exists\, x.\ Q\ x;\ (\mathit{SOME}\ x.\ P\ x) = (\mathit{SOME}\ x.\ Q\ x)]\!] \Longrightarrow \exists\, x.$
$P\ x \wedge Q\ x$
apply (*rule-tac* $x = (\mathit{SOME}\ x.\ P\ x)$ **in** *exI*)
apply (*rule conjI*)
apply (*blast intro*: *someI-ex*)
apply (*rule-tac* $s = (\mathit{SOME}\ x.\ Q\ x)$ **and** $t = (\mathit{SOME}\ x.\ P\ x)$ **in** *ssubst*, *assumption*)
apply (*blast intro*: *someI-ex*)
done

lemma *noIntersection-superset*: $[\![A \cap B = \{\};\ C \subseteq A]\!] \Longrightarrow C \cap B = \{\}$ **by** *blast*
lemma *diffImplSubset*: $A - B \subseteq A$ **by** *blast*
lemma *noIntersection-subsetDiff*: $[\![A \cap B = \{\};\ A \subseteq C]\!] \Longrightarrow A \subseteq C - B$ **by** *blast*

lemma *finiteMapUnion* [*elim*]: $[\![\mathit{finite}\ S;\ \bigwedge s.\ s \in S \Longrightarrow \mathit{finite}\ (f\ s)]\!] \Longrightarrow \mathit{finite}\ (\bigcup s \in S.\ f\ s)$
by *simp*

lemma *list-fixlen-expl*: $0 < \mathit{length}\ \mathit{xs} \Longrightarrow \mathit{xs} = (\mathit{hd}\ \mathit{xs})\ \#\ (\mathit{tl}\ \mathit{xs})$ **by** *force*

lemma *list-fixlen-expl1*: $\mathit{length}\ \mathit{xs} = 1 \Longrightarrow \mathit{xs} = [\mathit{hd}\ \mathit{xs}]$
apply (*subgoal-tac* $\mathit{length}\ \mathit{xs} = \mathit{Suc}\ 0$)
  prefer *2* **apply** *arith*

apply ($subgoal\text{-}tac\ \exists\, a\ as.\ xs = a\ \#\ as \wedge length\ as = 0$)
  prefer *2* **apply** (*clarsimp simp*: *length-Suc-conv*)
by *clarsimp*

lemma *list-fixlen-expl2*: $length\ xs = 2 \Longrightarrow xs = [hd\ xs,\ hd\ (tl\ xs)]$
apply ($subgoal\text{-}tac\ length\ xs = Suc\ (Suc\ 0)$)
  prefer *2* **apply** *arith*
apply ($subgoal\text{-}tac\ \exists\, a\ b\ bs.\ xs = a\ \#\ b\ \#\ bs \wedge length\ bs = 0$)
  prefer *2* **apply** (*clarsimp simp*: *length-Suc-conv*)
by *clarsimp*

lemma *semigroup-add.foldl-abelian-reverse*:
  $[\![semigroup\text{-}add\ add;\ \forall\, a\ b.\ add\ a\ b = add\ b\ a]\!] \Longrightarrow$
  $foldl\ add\ zero\ (xs) = foldl\ add\ zero\ (rev\ xs)$
apply (*induct-tac xs*, *simp*)
apply (*rename-tac x list*)
apply *simp*
apply (*erule-tac* $s = foldl\ add\ zero\ list$ **in** *subst*)
by (*rule semigroup-add.foldl-assoc*, *assumption*)

lemma *predicate-true-if-mem*: $x \in S \Longrightarrow S\ x$ **by** (*simp add*: *mem-def*)
lemma *mem-if-predicate-true*: $S\ x \Longrightarrow x \in S$ **by** (*simp add*: *mem-def*)

lemma *predicate-if-in-lambda*: $x \in (\lambda x.\ P\ x) \Longrightarrow P\ x$ **by** (*simp add*: *mem-def*)

lemma *set-ops-to-predicate.simps*: **shows**
  $S\ x \Longrightarrow (S \cup T)\ x$ **and**
  $T\ x \Longrightarrow (S \cup T)\ x$ **and**
  $[\![S\ x;\ T\ x]\!] \Longrightarrow (S \cap T)\ x$ **and**
  $x = y \Longrightarrow (insert\ y\ S)\ x$ **and**
  $S\ x \Longrightarrow (insert\ y\ S)\ x$
by (*blast intro*: *predicate-true-if-mem mem-if-predicate-true*)+

definition *powermultiset* :: $'a\ set \Rightarrow ('a\ multiset)set$
where $powermultiset\ S \equiv \{M.\ set\text{-}of\ M \subseteq S\}$

primrec *list-times* :: $('a\ set)list \Rightarrow ('a\ list)set$ **where**
$list\text{-}times\ [] = \{[]\}\ |$
$list\text{-}times\ (x\ \#\ xs) = \{l.\ hd\ l \in x \wedge tl\ l \in list\text{-}times\ xs \wedge length\ l = Suc\ (length\ xs)\}$

primrec *list-times-compr* :: $('a)list \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow ('b\ list)set$ **where**
$list\text{-}times\text{-}compr\ []\ f = \{[]\}\ |$
$list\text{-}times\text{-}compr\ (x\ \#\ xs)\ f =$
  $\{l.\ hd\ l \in f\ x \wedge tl\ l \in list\text{-}times\text{-}compr\ xs\ f \wedge length\ l = Suc\ (length\ xs)\}$

definition *multiset-of* :: $'a\ set \Rightarrow 'a\ multiset$ **where**
$multiset\text{-}of\ S \equiv Abs\text{-}multiset\ (\lambda x.\ if\ x \in S\ then\ 1\ else\ 0)$

```
end

theory PetriNet
imports Main Multiset DrahflowTools
begin

types ('e,'act)petrinet-repr =
  ('e set)×('e set)×('e × 'e)set×('e ⇒ 'act)×('e set)×('act set)

definition wellformed-petrinet :: ('e,'act)petrinet-repr ⇒ bool where
  wellformed-petrinet N ≡
    let (S, T, F, l, M0, τSet) = N in (
    (∀ s x. (s, x) ∈ F ∧ s ∈ S ⟶ x ∈ T) ∧
    (∀ t x. (t, x) ∈ F ∧ t ∈ T ⟶ x ∈ S) ∧
    ¬(∃ x. x ∈ S ∩ T) ∧
    (∀ s. s ∈ M0 ⟶ s ∈ S)
    )

typedef ('e,'act)petrinet =
  {N :: ('e,'act)petrinet-repr. wellformed-petrinet N}
apply (rule exI[where x = ({s}, {}, {}, (λs. a), {}, {})])
by (simp add: CollectI wellformed-petrinet-def Let-def)

definition places :: ('e,'act)petrinet ⇒ 'e set
  where places N ≡ fst (Rep-petrinet N)
definition transitions :: ('e,'act)petrinet ⇒ 'e set
  where transitions N ≡ fst (snd (Rep-petrinet N))
definition label :: ('e,'act)petrinet ⇒ ('e ⇒ 'act)
  where label N ≡ fst (snd (snd (snd (Rep-petrinet N))))
definition flow :: ('e,'act)petrinet ⇒ ('e×'e) set
  where flow N ≡ fst (snd (snd (Rep-petrinet N)))
definition initial :: ('e,'act)petrinet ⇒ ('e set)
  where initial N ≡ fst (snd (snd (snd (snd (Rep-petrinet N)))))
definition silent ::
    ('e,'act)petrinet ⇒ 'act set
  where silent N ≡ snd (snd (snd (snd (snd (Rep-petrinet N)))))
definition static ::
    ('e,'act)petrinet ⇒ ('e set)×('e set)×(('e×'e)set)×('e ⇒ 'act)
  where
    static N ≡ let (S, T, F, l, M0, τSet) = Rep-petrinet N in (S, T, F, l)

definition Net ::
    ('e set)×('e set)×(('e×'e)set)×('e ⇒ 'act)×('e set)×('act set) ⇒
       ('e,'act)petrinet
  where [simp]: Net tuple = Abs-petrinet tuple
```

definition *preset* ::
  $('e,'act)petrinet \Rightarrow 'e \Rightarrow 'e\ set$
  where $preset\ N\ x \equiv \{y.\ (y,\ x) \in flow\ N\}$
definition *postset* ::
  $('e,'act)\ petrinet \Rightarrow 'e \Rightarrow 'e\ set$
  where $postset\ N\ x \equiv \{y.\ (x,\ y) \in flow\ N\}$

definition *presetSet* ::
  $('e,'act)petrinet \Rightarrow 'e\ set \Rightarrow 'e\ set$
  where $presetSet\ N\ X \equiv \{y.\ \exists x \in X.\ (y,\ x) \in flow\ N\}$
definition *postsetSet* ::
  $('e,'act)petrinet \Rightarrow 'e\ set \Rightarrow 'e\ set$
  where $postsetSet\ N\ X \equiv \{y.\ \exists x \in X.\ (x,\ y) \in flow\ N\}$

definition *step* ::
  $('e,'act)petrinet \Rightarrow 'e\ set \Rightarrow 'e\ set \Rightarrow 'e\ set \Rightarrow bool$
  where
  $step\ N\ M_1\ G\ M_2 \equiv$
  $(G \subseteq transitions\ N) \wedge G \neq \{\} \wedge$
  $(\forall t \in G.\ preset\ N\ t \subseteq M_1 \wedge (M_1 - preset\ N\ t) \cap postset\ N\ t = \{\}) \wedge$
  $(\forall t \in G.\ \forall u \in G.\ t \neq u \longrightarrow$
  $preset\ N\ t \cap preset\ N\ u = \{\} \wedge postset\ N\ t \cap postset\ N\ u = \{\}) \wedge$
  $(M_2 = (M_1 - presetSet\ N\ G) \cup postsetSet\ N\ G)$

inductive-set $reachable :: ('e,'act)petrinet \Rightarrow ('e\ set)set$
  for $N :: ('e,'act)petrinet$ **where**
  $reachable\text{-}start\text{: } initial\ N \in reachable\ N$
| $reachable\text{-}step\text{: } [\![M_1 \in reachable\ N;\ \exists G.\ step\ N\ M_1\ G\ M_2]\!] \Longrightarrow M_2 \in reachable\ N$

definition $plain :: ('e,'act)petrinet \Rightarrow bool$ **where**
  $plain\ N \equiv \forall t \in transitions\ N.\ label\ N\ t \notin silent\ N \wedge$
  $(\forall u \in transitions\ N.\ (label\ N\ t = label\ N\ u) \longrightarrow (t = u))$

definition $\tau Plain :: ('e,'act)petrinet \Rightarrow bool$ **where**
  $\tau Plain\ N \equiv \forall t \in transitions\ N.\ \forall u \in transitions\ N.$
  $label\ N\ t = label\ N\ u \longrightarrow$
  $(silent\ N\ (label\ N\ t)) \vee$
  $(silent\ N\ (label\ N\ u)) \vee$
  $(t = u)$

definition $contactFree :: ('e,'act)petrinet \Rightarrow bool$ **where**
  $contactFree\ N \equiv$
  $\forall M \in reachable\ N.\ \forall t \in transitions\ N.\ preset\ N\ t \subseteq M \longrightarrow$
  $(M - preset\ N\ t) \cap postset\ N\ t = \{\}$

definition *contactFreeStep* ::
  $('e,'act)petrinet \Rightarrow 'e\ set \Rightarrow 'e\ set \Rightarrow 'e\ set \Rightarrow bool$

where
  contactFreeStep $N\ M_1\ G\ M_2 \equiv$
    $(G \subseteq \mathit{transitions}\ N) \wedge G \neq \{\} \wedge$
    $(\forall\, t \in G.\ \mathit{preset}\ N\ t \subseteq M_1) \wedge$
    $(\forall\, t \in G.\ \forall\, u \in G.\ t \neq u \longrightarrow \mathit{preset}\ N\ t \cap \mathit{preset}\ N\ u = \{\}) \wedge$
    $(M_2 = (M_1 - \mathit{presetSet}\ N\ G) \cup \mathit{postsetSet}\ N\ G)$

lemma *stepImplContactFreeStep*: $[\![\mathit{step}\ N\ M_1\ G\ M_2]\!] \Longrightarrow \mathit{contactFreeStep}\ N\ M_1\ G\ M_2$
by (*simp add*: *contactFreeStep-def step-def*)

lemma *contactFreeStep-lemma1*:
$[\![(M - \mathit{PreT}) \cap \mathit{PostT} = \{\};\ \mathit{PreT} \subseteq M;\ \mathit{PreU} \subseteq M;\ \mathit{PreT} \cap \mathit{PreU} = \{\}]\!] \Longrightarrow$
$\mathit{PostT} \cap \mathit{PreU} = \{\}$
by *blast*

lemma *contactFreeStepValid*:
$[\![\mathit{contactFree}\ N;\ M_1 \in \mathit{reachable}\ N;\ \mathit{contactFreeStep}\ N\ M_1\ G\ M_2]\!] \Longrightarrow \mathit{step}\ N\ M_1\ G\ M_2$
apply (*unfold contactFree-def*, *unfold contactFreeStep-def*)
apply (*subgoal-tac* $\forall\, t \in \mathit{transitions}\ N.\ \mathit{preset}\ N\ t \subseteq M_1 \longrightarrow$
    $(M_1 - \mathit{preset}\ N\ t) \cap \mathit{postset}\ N\ t = \{\}$)
  prefer *2* **apply** *blast*
apply (*unfold step-def*)
apply (*rule conjI*, *blast*)
apply (*rule conjI*, *blast*)
apply (*rule conjI*, *rule ballI*)
  apply (*rule conjI*, *blast*)
  apply (*erule conjE*)+
  apply (*subgoal-tac* $t \in \mathit{transitions}\ N$, *simp*)
  apply (*rule set-mp*[**where** $A = G$], *assumption*+)
apply (*rule conjI*)
  prefer *2* **apply** *simp*
apply (*rule ballI*)+
apply (*rule impI*, *rule conjI*, *simp*)

The interesting part of the proof follows.

apply (*erule conjE*)+
apply (*subgoal-tac step* $N\ M_1\ \{t\}\ (M_1 - \mathit{preset}\ N\ t \cup \mathit{postset}\ N\ t)$)
  prefer *2*
  apply (*unfold step-def*)[*1*]
  apply (*rule conjI*, *blast*)
  apply (*rule conjI*, *blast*)
  apply (*rule conjI*, *simp*)
    apply (*erule ballE-in*[**where** $x = M_1$], *assumption*)
    apply (*subgoal-tac* $t \in \mathit{transitions}\ N$, *simp*)
    apply (*rule set-mp*[**where** $B = \mathit{transitions}\ N$ **and** $A = G$], *assumption*+)
  apply (*simp add*: *presetSet-def postsetSet-def preset-def postset-def*)

apply (*subgoal-tac* (M_1 − *preset* N t ∪ *postset* N t) ∈ *reachable* N)
  prefer *2*
  apply (*rule reachable-step*[**where** $M_1 = M_1$], *assumption*)
  apply *blast*

apply (*thin-tac* $G \neq \{\}$)
apply (*thin-tac step* N M_1 $\{t\}$ (M_1 − *preset* N t ∪ *postset* N t))
apply (*thin-tac* $M_2 = M_1$ − *presetSet* N G ∪ *postsetSet* N G)

apply (*subgoal-tac* ((M_1 − *preset* N t ∪ *postset* N t) − *preset* N u) ∩ *postset* N u = {})
  apply (*rule-tac* A = (M_1 − *preset* N t ∪ *postset* N t) − *preset* N u **and** B = *postset* N u
    and C = *postset* N t **in** *noIntersection-superset*, *assumption*)
  apply (*subgoal-tac postset* N t ∩ *preset* N u = {})
   apply (*rule-tac* A = *postset* N t **and** B = *preset* N u **in** *noIntersection-subsetDiff*,
    assumption)
   apply *blast*
  apply (*subgoal-tac preset* N u ⊆ M_1 − *preset* N t)
   apply (*erule-tac* $x = t$ **and** $Q = \lambda t.\ \forall u{\in}G.\ t \neq u \longrightarrow$ *preset* N t ∩ *preset* N u = {}
    in *ballE-in*, *assumption*)
   apply (*erule-tac* $x = u$ **and** $Q = \lambda u.\ t \neq u \longrightarrow$ *preset* N t ∩ *preset* N u = {}
    in *ballE-in*, *assumption*)
   apply (*erule impE*, *assumption*)
   apply (*rule-tac* $M = M_1$ **and** $PreT$ = *preset* N t **in** *contactFreeStep-lemma1*)
      apply (*erule-tac* $x = t$ **and** A = *transitions* N **in** *ballE-in*, *blast*)
      apply (*erule-tac* $x = t$ **and** $A = G$ **in** *ballE-in*, *blast*)
      apply *blast*
     apply (*erule-tac* $x = t$ **and** $A = G$ **in** *ballE-in*, *simp*)
     apply *assumption*
    apply (*erule-tac* $x = u$ **and** $A = G$ **in** *ballE-in*, *simp*)
    apply *assumption*
   apply *assumption*
  apply (*thin-tac* M_1 ∈ *reachable* N)
  apply (*thin-tac* ∀ M∈*reachable* N. ∀ t∈*transitions* N. *preset* N t ⊆ M ⟶
      (M − *preset* N t) ∩ *postset* N t = {})
  apply (*thin-tac* ∀ t∈*transitions* N. *preset* N t ⊆ M_1 ⟶
      (M_1 − *preset* N t) ∩ *postset* N t = {})
  apply *blast*

apply (*subgoal-tac preset* N u ⊆ (M_1 − *preset* N t ∪ *postset* N t))
  apply (*erule-tac* x = (M_1 − *preset* N t ∪ *postset* N t) **in** *ballE-in*, *assumption*)
  apply (*erule-tac* $x = u$ **and**
        $Q = \lambda u.$ *preset* N u ⊆ M_1 − *preset* N t ∪ *postset* N t ⟶
         ((M_1 − *preset* N t ∪ *postset* N t) − *preset* N u) ∩ *postset* N u = {}
     in *ballE-in*, *blast*)
  apply (*erule impE*, *assumption*)
  apply *assumption*

apply ($thin\text{-}tac\ \forall M{\in}reachable\ N.\ \forall t{\in}transitions\ N.\ preset\ N\ t \subseteq M \longrightarrow$
  $(M - preset\ N\ t) \cap postset\ N\ t = \{\}$)
apply ($thin\text{-}tac\ \forall t{\in}transitions\ N.\ preset\ N\ t \subseteq M_1 \longrightarrow (M_1 - preset\ N\ t) \cap postset\ N\ t = \{\}$)
apply ($erule\text{-}tac\ x = t$ **and** $Q = \lambda t.\ \forall u{\in}G.\ t \neq u \longrightarrow preset\ N\ t \cap preset\ N\ u = \{\}$
 in $ballE\text{-}in$, $assumption$)
apply ($erule\text{-}tac\ x = u$ **and** $Q = \lambda u.\ t \neq u \longrightarrow preset\ N\ t \cap preset\ N\ u = \{\}$
 in $ballE\text{-}in$, $assumption$)
by $blast$

lemma $contactFreeStepEquiv$:
 $[\![contactFree\ N;\ M_1 \in reachable\ N]\!] \Longrightarrow step\ N\ M_1\ G\ M_2 = contactFreeStep\ N\ M_1\ G\ M_2$
by ($rule\ iffI$, $simp\ add$: $stepImplContactFreeStep$, $simp\ add$: $contactFreeStepValid$)

definition $finitelyMarked :: ('e,'act)petrinet \Rightarrow bool$ **where**
 $finitelyMarked\ N \equiv$
  $finite\ (initial\ N) \wedge$
  $(\forall t \in transitions\ N.\ \exists s \in places\ N.\ (s,\ t) \in flow\ N) \wedge$
  $(\forall t \in transitions\ N.\ finite\ (postset\ N\ t))$

lemma $finiteStepImplFinitePostSet$ [$intro$]:
 $[\![\forall t \in G.\ finite\ (postset\ N\ t);\ finite\ G;\ G \subseteq transitions\ N]\!] \Longrightarrow finite\ (postsetSet\ N\ G)$
apply ($simp\ add$: $postsetSet\text{-}def$)
apply ($subgoal\text{-}tac\ \{y.\ \exists x{\in}G.\ (x,\ y) \in flow\ N\} = (\bigcup t \in G.\ postset\ N\ t)$)
 prefer 2 **apply** ($simp\ add$: $postset\text{-}def$, $blast$)
apply ($erule\text{-}tac\ s = (\bigcup t \in G.\ postset\ N\ t)$ **and** $t = \{y.\ \exists x{\in}G.\ (x,\ y) \in flow\ N\}$ **in** $ssubst$)
by $simp$

lemma $finitelyMarkedEverywhere$: $[\![finitelyMarked\ N;\ M \in reachable\ N]\!] \Longrightarrow finite\ M$
apply ($unfold\ finitelyMarked\text{-}def$)
apply ($erule\ reachable.induct$, $simp$)
apply ($erule\ exE$)
apply ($subgoal\text{-}tac\ finite\ G$)
 apply ($simp\ add$: $step\text{-}def$)
 apply ($erule\ conjE$)+
 apply ($rule\text{-}tac\ N = N$ **and** $G = G$ **in** $finiteStepImplFinitePostSet$)
   apply $blast$
  apply $assumption$
 apply $assumption$

apply ($simp\ add$: $step\text{-}def$)
apply ($erule\ conjE$)+
apply ($thin\text{-}tac\ M_1 \in reachable\ N$)
apply ($thin\text{-}tac\ finite\ (initial\ N)$)
apply ($thin\text{-}tac\ M_2 = M_1 - presetSet\ N\ G \cup postsetSet\ N\ G$)
apply ($thin\text{-}tac\ G \neq \{\}$)
apply ($thin\text{-}tac\ \forall t{\in}transitions\ N.\ finite\ (postset\ N\ t)$)

apply (*rule-tac* $f = \lambda t.\ SOME\ s.\ (s,\ t) \in flow\ N$ **in** *finite-imageD*)
  apply (*subgoal-tac* $((\lambda t.\ SOME\ s.\ (s,\ t) \in flow\ N)$ ‘ $G) \subseteq M_1$)
    apply (*erule-tac* $A = (\lambda t.\ SOME\ s.\ (s,\ t) \in flow\ N)$ ‘ G **and** $B = M_1$ **in** *finite-subset*)
    apply *assumption*
  apply (*thin-tac finite* M_1)
  apply (*rule subsetI*)
  apply *clarify*
  apply (*erule-tac* $A = G$ **and** $x = t$ **in** *ballE-in*, *assumption*)
  apply (*erule conjE*)+
  apply (*simp add*: *preset-def*)
  apply (*rule-tac* $Q = \lambda x.\ x \in M_1$ **in** *someI2-ex*)
    apply (*erule-tac* $x = t$ **and** $A = transitions\ N$ **in** *ballE-in*, *blast*)
    apply *blast*
  apply *blast*

apply (*rule inj-onI*, *rename-tac t u*)
apply (*rule directContradiction*)
apply (*erule-tac* $x = t$ **and** $y = u$ **in** *ballE-in-double*, *assumption*+)
apply (*erule impE*, *assumption*, *erule conjE*)
apply (*frule-tac* $x = t$ **and** $A = transitions\ N$ **and** $P = \lambda t.\ \exists s \in places\ N.\ (s,\ t) \in flow\ N$
  in *ballE-in*, *blast*, *assumption*)
apply (*frule-tac* $x = u$ **and** $A = transitions\ N$ **and** $P = \lambda u.\ \exists s \in places\ N.\ (s,\ u) \in flow\ N$
  in *ballE-in*, *blast*, *assumption*)
apply (*frule-tac* $A = places\ N$ **and** $P = \lambda s.\ (s,\ t) \in flow\ N$ **in** *bexToEx*)
apply (*frule-tac* $A = places\ N$ **and** $P = \lambda s.\ (s,\ u) \in flow\ N$ **in** *bexToEx*)
apply (*frule-tac* $P = \lambda s.\ (s,\ t) \in flow\ N$ **and** $Q = \lambda s.\ (s,\ u) \in flow\ N$
  in *some-connect*, *assumption*+)
apply (*simp add*: *preset-def*)
by *blast*

definition *distributed* $N \equiv$
  $\exists coloc.\ (\forall t \in transitions\ N.\ \forall s \in preset\ N\ t.\ coloc\ s\ t) \wedge$
    $(\forall t\ u\ M_1\ G\ M_2.\ (reachable\ N\ M_1 \wedge t \in G \wedge u \in G \wedge step\ N\ M_1\ G\ M_2) \longrightarrow \neg\ coloc\ t\ u)$

lemma *distributed-by-mapping*:
  $\exists loc.\ (\forall t \in transitions\ N.\ \forall s \in preset\ N\ t.\ loc\ s = loc\ t) \wedge$
    $(\forall t\ u\ M_1\ G\ M_2.\ (reachable\ N\ M_1 \wedge t \in G \wedge u \in G \wedge step\ N\ M_1\ G\ M_2) \longrightarrow$
      $loc\ t \neq loc\ u) \Longrightarrow distributed\ N$
apply (*simp add*: *distributed-def*)
by (*erule exE*, *rule-tac* $x = \lambda x\ y.\ loc\ x = loc\ y$ **in** *exI*)

definition *stepTraces* $N \equiv$
  $\{Trace.\ \exists Gs\ Ms.\ foldl\ (\lambda t\ (M_1,\ G,\ M_2).\ t \wedge step\ N\ M_1\ G\ M_2)\ True$
      $(zip\ (initial\ N\ \#\ Ms)\ (zip\ Gs\ Ms)) \wedge$
    $Trace = map\ (\lambda G.\ Abs\text{-}multiset\ (\lambda a.\ card\ \{t \in G.\ label\ N\ t = a \wedge a \notin silent\ N\}))\ Gs\}$

definition *plainify* :: $('e,'act)petrinet \Rightarrow ('e,'e)petrinet$

where *plainify N* $\equiv$ *Abs-petrinet* ((*places N*), (*transitions N*), (*flow N*), *id*, (*initial N*), {})

lemma *petrinet.access*:
shows $[\![(S,\ T,\ F,\ l,\ M_0,\ \tau Set) \in petrinet]\!] \Longrightarrow$
  $places\ (Abs\text{-}petrinet\ (S,\ T,\ F,\ l,\ M_0,\ \tau Set)) = S$
 and $[\![(S,\ T,\ F,\ l,\ M_0,\ \tau Set) \in petrinet]\!] \Longrightarrow$
  $transitions\ (Abs\text{-}petrinet\ (S,\ T,\ F,\ l,\ M_0,\ \tau Set)) = T$
 and $[\![(S,\ T,\ F,\ l,\ M_0,\ \tau Set) \in petrinet]\!] \Longrightarrow$
  $flow\ (Abs\text{-}petrinet\ (S,\ T,\ F,\ l,\ M_0,\ \tau Set)) = F$
 and $[\![(S,\ T,\ F,\ l,\ M_0,\ \tau Set) \in petrinet]\!] \Longrightarrow$
  $label\ (Abs\text{-}petrinet\ (S,\ T,\ F,\ l,\ M_0,\ \tau Set)) = l$
 and $[\![(S,\ T,\ F,\ l,\ M_0,\ \tau Set) \in petrinet]\!] \Longrightarrow$
  $initial\ (Abs\text{-}petrinet\ (S,\ T,\ F,\ l,\ M_0,\ \tau Set)) = M_0$
 and $[\![(S,\ T,\ F,\ l,\ M_0,\ \tau Set) \in petrinet]\!] \Longrightarrow$
  $silent\ (Abs\text{-}petrinet\ (S,\ T,\ F,\ l,\ M_0,\ \tau Set)) = \tau Set$
by (
 (*simp add*: *places-def transitions-def flow-def label-def initial-def silent-def*),
 (*subgoal-tac Rep-petrinet* $(Abs\text{-}petrinet\ (S,\ T,\ F,\ l,\ M_0,\ \tau Set)) = (S,\ T,\ F,\ l,\ M_0,\ \tau Set)$,
  simp),
 (*blast intro*: *Abs-petrinet-inverse*)
)+

lemma *petrinet.unfold-raw*:
 $[\![(S,\ T,\ F,\ l,\ M_0,\ \tau Set) = Rep\text{-}petrinet\ N;$
  $(S,\ T,\ F,\ l,\ M_0,\ \tau Set) \in petrinet \Longrightarrow P\ (Abs\text{-}petrinet\ (S,\ T,\ F,\ l,\ M_0,\ \tau Set))]\!] \Longrightarrow\ P\ N$
apply (*subgoal-tac* $(S,\ T,\ F,\ l,\ M_0,\ \tau Set) \in petrinet$)
 apply (*subgoal-tac* $P\ (Abs\text{-}petrinet\ (S,\ T,\ F,\ l,\ M_0,\ \tau Set))$)
  apply (*simp add*: *Rep-petrinet-inverse*)
 apply *blast*
by (*erule ssubst*, *rule Rep-petrinet*)

lemma *petrinet.unfold*:
$[\![(places\ N,\ transitions\ N,\ flow\ N,\ label\ N,\ initial\ N,\ silent\ N) \in petrinet \Longrightarrow$
  $P\ (Abs\text{-}petrinet\ (places\ N,\ transitions\ N,\ flow\ N,\ label\ N,\ initial\ N,\ silent\ N))]\!]$
$\Longrightarrow P\ N$
apply (*rule-tac* $S = places\ N$ **and** $T = transitions\ N$ **and** $F = flow\ N$ **and** $l = label\ N$
  and $M_0 = initial\ N$ **and** $\tau Set = silent\ N$ **in** *petrinet.unfold-raw*)
 apply (*simp add*: *petrinet-def places-def transitions-def flow-def label-def initial-def silent-def*)
by *blast*

lemma *plainify-successful-raw* [*intro*!]:
 $[\![(S,\ T,\ F,\ l,\ M_0,\ \tau Set) \in petrinet]\!] \Longrightarrow plain\ (plainify\ (Abs\text{-}petrinet\ (S,\ T,\ F,\ l,\ M_0,\ \tau Set)))$
apply (*simp add*: *plain-def plainify-def*)
apply (*subgoal-tac* $(S,\ T,\ F,\ id,\ M_0,\ \{\}) \in petrinet$, *simp add*: *petrinet.access*)
apply (*simp add*: *petrinet-def*)
apply (*unfold wellformed-petrinet-def*)
apply (*simp only*: *Let-def*)

by *blast*

lemma *plainify-successful* [*intro*!]: *plain* (*plainify N*)
apply (*rule-tac N* = *N* **in** *petrinet.unfold*)
by *blast*

end

theory *AsynFSM*
imports *Main Multiset DrahflowTools*
begin

typedef ('*act*)*actsig* =
  {Σ :: ('*act set*)×('*act set*)×('*act set*).
    let (Σi, Σo, $\Sigma\tau$) = Σ *in* $\Sigma i \cap \Sigma o = \{\} \wedge \Sigma i \cap \Sigma\tau = \{\} \wedge \Sigma o \cap \Sigma\tau = \{\}$}
apply (*rule exI*[**where** *x* = ({}, {}, {})])
by *simp*

definition *input* :: ('*act*)*actsig* ⇒ '*act set* **where** *input* $\Sigma \equiv$ *fst* (*Rep-actsig* Σ)
definition *output* :: ('*act*)*actsig* ⇒ '*act set* **where** *output* $\Sigma \equiv$ *fst* (*snd* (*Rep-actsig* Σ))
definition *inner* :: ('*act*)*actsig* ⇒ '*act set* **where** *inner* $\Sigma \equiv$ *snd* (*snd* (*Rep-actsig* Σ))

lemma *actsig.unfold-raw*:
  ⟦(*In*, *Out*, *Inner*) = *Rep-actsig* Σ;
    (*In*, *Out*, *Inner*) ∈ *actsig* ⟹ *P* (*Abs-actsig* (*In*, *Out*, *Inner*))⟧ ⟹ *P* Σ
apply (*subgoal-tac* (*In*, *Out*, *Inner*) ∈ *actsig*)
  apply (*subgoal-tac P* (*Abs-actsig* (*In*, *Out*, *Inner*)))
    apply (*simp add*: *Rep-actsig-inverse*)
  apply *blast*
by (*erule ssubst*, *rule Rep-actsig*)

lemma *actsig.unfold*:
⟦(*input* Σ, *output* Σ, *inner* Σ) ∈ *actsig* ⟹ *P* (*Abs-actsig* (*input* Σ, *output* Σ, *inner* Σ))⟧
⟹ *P* Σ
apply (*rule-tac In* = *input* Σ **and** *Out* = *output* Σ **and** *Inner* = *inner* Σ **in** *actsig.unfold-raw*)
  apply (*simp add*: *actsig-def input-def output-def inner-def*)
by *blast*

lemma *input-access* [*simp*]:
  (*In*, *Out*, *Inner*) ∈ *actsig* ⟹ *input* (*Abs-actsig* (*In*, *Out*, *Inner*)) = *In*
apply (*simp add*: *input-def*)
apply (*subgoal-tac Rep-actsig* (*Abs-actsig* (*In*, *Out*, *Inner*)) = (*In*, *Out*, *Inner*), *simp*)
by (*blast intro*: *Abs-actsig-inverse*)

lemma *output-access* [*simp*]:
  (*In*, *Out*, *Inner*) ∈ *actsig* ⟹ *output* (*Abs-actsig* (*In*, *Out*, *Inner*)) = *Out*
apply (*simp add*: *output-def*)

apply (*subgoal-tac Rep-actsig* (*Abs-actsig* (*In*, *Out*, *Inner*)) = (*In*, *Out*, *Inner*), *simp*)
by (*blast intro*: *Abs-actsig-inverse*)

lemma *inner-access* [*simp*]:
  (*In*, *Out*, *Inner*) $\in$ *actsig* $\Longrightarrow$ *inner* (*Abs-actsig* (*In*, *Out*, *Inner*)) = *Inner*
apply (*simp add*: *inner-def*)
apply (*subgoal-tac Rep-actsig* (*Abs-actsig* (*In*, *Out*, *Inner*)) = (*In*, *Out*, *Inner*), *simp*)
by (*blast intro*: *Abs-actsig-inverse*)

typedef ($'q$,$'act$)*asynfsm* =
  {*A* :: ($'act$ *actsig*)$\times$($'q$ *set*)$\times$($'q$)$\times$(($'q\times$($'act$ *set*)$\times$($'act$ *set*)$\times'q$)*set*).
    let (Σ, Q, q_0, *stepRel*) = *A in* ($q_0 \in Q \wedge$ ($\forall$ (q, *In*, *Out*, q') $\in$ *stepRel*. *In* $\neq$ {} $\wedge$
      $q \in Q \wedge q' \in Q \wedge$ *Out* $\subseteq$ *output* Σ $\wedge$ *In* $\subseteq$ (*input* Σ $\cup$ *inner* Σ)))}
apply (*rule exI*[**where** x = *let* q = (*SOME* x. *True*) *in* (*Abs-actsig* ({}, {}, {}), {q}, q, {})])
by (*simp add*: *Let-def*)

definition *actions* :: ($'q$,$'act$)*asynfsm* $\Rightarrow$ ($'act$ *actsig*) **where** *actions A* $\equiv$ *fst* (*Rep-asynfsm A*)
definition *states* :: ($'q$,$'act$)*asynfsm* $\Rightarrow$ ($'q$ *set*) **where** *states A* $\equiv$ *fst* (*snd* (*Rep-asynfsm A*))
definition *initial* :: ($'q$,$'act$)*asynfsm* $\Rightarrow$ $'q$ **where** *initial A* $\equiv$ *fst* (*snd* (*snd* (*Rep-asynfsm A*)))
definition *steps* :: ($'q$,$'act$)*asynfsm* $\Rightarrow$ (($'q\times$($'act$ *set*)$\times$($'act$ *set*)$\times'q$)*set*)
  where *steps A* = *snd* (*snd* (*snd* (*Rep-asynfsm A*)))

lemma *asynfsm.unfold-raw*:
  $[\![$(Σ, Q, q_0, *stepRel*) = *Rep-asynfsm A*;
    (Σ, Q, q_0, *stepRel*) $\in$ *asynfsm* $\Longrightarrow$ *P* (*Abs-asynfsm* (Σ, Q, q_0, *stepRel*))$]\!]$ $\Longrightarrow$ *P A*
apply (*subgoal-tac* (Σ, Q, q_0, *stepRel*) $\in$ *asynfsm*)
 apply (*subgoal-tac P* (*Abs-asynfsm* (Σ, Q, q_0, *stepRel*)))
  apply (*simp add*: *Rep-asynfsm-inverse*)
 apply *blast*
by (*erule ssubst*, *rule Rep-asynfsm*)

lemma *asynfsm.unfold*:
  $[\![$(*actions A*, *states A*, *initial A*, *steps A*) $\in$ *asynfsm* $\Longrightarrow$
    P (*Abs-asynfsm* (*actions A*, *states A*, *initial A*, *steps A*))$]\!]$ $\Longrightarrow$ *P A*
apply (*rule-tac* Σ = *actions A* **and** Q = *states A* **and** q_0 = *initial A*
  and *stepRel* = *steps A* **in** *asynfsm.unfold-raw*)
 apply (*simp add*: *asynfsm-def actions-def states-def initial-def steps-def*)
by *blast*

definition *step* :: ($'q$,$'act$)*asynfsm* $\Rightarrow$ $'q$ $\Rightarrow$ ($'act$ *set*) $\Rightarrow$ ($'act$ *set*) $\Rightarrow$ $'q$ $\Rightarrow$ *bool*
  where *step A q In Out* q' $\equiv$ (q, *In*, *Out*, q') $\in$ *steps A*

lemma *actions-access* [*simp*]:
  (Σ, Q, q_0, *stepRel*) $\in$ *asynfsm* $\Longrightarrow$ *actions* (*Abs-asynfsm* (Σ, Q, q_0, *stepRel*)) = Σ
apply (*simp add*: *actions-def*)
apply (*subgoal-tac Rep-asynfsm* (*Abs-asynfsm* (Σ, Q, q_0, *stepRel*)) = (Σ, Q, q_0, *stepRel*), *simp*)
by (*blast intro*: *Abs-asynfsm-inverse*)

lemma *states-access* [*simp*]:
  $(\Sigma,\ Q,\ q_0,\ stepRel) \in asynfsm \Longrightarrow states\ (Abs\text{-}asynfsm\ (\Sigma,\ Q,\ q_0,\ stepRel)) = Q$
apply (*simp add*: *states-def*)
apply (*subgoal-tac Rep-asynfsm* (*Abs-asynfsm* $(\Sigma, Q, q_0, stepRel)) = (\Sigma, Q, q_0, stepRel)$, *simp*)
by (*blast intro*: *Abs-asynfsm-inverse*)

lemma *initial-access* [*simp*]:
  $(\Sigma,\ Q,\ q_0,\ stepRel) \in asynfsm \Longrightarrow initial\ (Abs\text{-}asynfsm\ (\Sigma,\ Q,\ q_0,\ stepRel)) = q_0$
apply (*simp add*: *initial-def*)
apply (*subgoal-tac Rep-asynfsm* (*Abs-asynfsm* $(\Sigma, Q, q_0, stepRel)) = (\Sigma, Q, q_0, stepRel)$, *simp*)
by (*blast intro*: *Abs-asynfsm-inverse*)

lemma *steps-access* [*simp*]:
  $(\Sigma,\ Q,\ q_0,\ stepRel) \in asynfsm \Longrightarrow steps\ (Abs\text{-}asynfsm\ (\Sigma,\ Q,\ q_0,\ stepRel)) = stepRel$
apply (*simp add*: *steps-def*)
apply (*subgoal-tac Rep-asynfsm* (*Abs-asynfsm* $(\Sigma, Q, q_0, stepRel)) = (\Sigma, Q, q_0, stepRel)$, *simp*)
by (*blast intro*: *Abs-asynfsm-inverse*)

lemma *initial-in-states* [*intro*]: $initial\ A \in states\ A$
apply (*simp add*: *initial-def states-def*)
apply (*subgoal-tac Rep-asynfsm* $A \in asynfsm$)
  prefer *2* **apply** (*rule Rep-asynfsm*)
by (*clarsimp simp*: *asynfsm-def*)

lemma *nothing-in-emptyset*: $A = \{\} \Longrightarrow y \notin A$ **by** *blast*

lemma *step-respects-signature* [*rule-format*]:
shows $step\ A\ q_1\ In\ Out\ q_2 \longrightarrow Out \subseteq output\ (actions\ A)$
and $step\ A\ q_1\ In\ Out\ q_2 \longrightarrow In \subseteq (input\ (actions\ A)) \cup (inner\ (actions\ A))$
and $step\ A\ q_1\ In\ Out\ q_2 \longrightarrow In \cap output\ (actions\ A) = \{\}$
and $step\ A\ q_1\ In\ Out\ q_2 \longrightarrow Out \cap ((input\ (actions\ A)) \cup (inner\ (actions\ A))) = \{\}$
apply *succeed*
      apply (*rule asynfsm.unfold*)
      apply (*clarsimp simp*: *step-def steps-access actions-access*)
      apply (*unfold asynfsm-def*, *clarsimp*)
      apply (*erule-tac* $x = (q_1, In, Out, q_2)$ **in** *ballE-in*, *assumption*, *blast*)
    apply (*rule asynfsm.unfold*)
    apply (*clarsimp simp*: *step-def steps-access actions-access*)
    apply (*unfold asynfsm-def*, *clarsimp*)
    apply (*erule-tac* $x = (q_1, In, Out, q_2)$ **in** *ballE-in*, *assumption*, *blast*)
  apply (*rule asynfsm.unfold*)
  apply (*clarsimp simp*: *step-def steps-access actions-access*)
  apply (*unfold asynfsm-def*, *clarsimp*)
  apply (*erule-tac* $x = (q_1, In, Out, q_2)$ **in** *ballE-in*, *assumption*, *clarsimp*)
  apply (*rule actsig.unfold*)
  apply (*subst output-access*, *assumption*)

apply (*unfold actsig-def*)
apply (*clarsimp*, *rule equals0I*, (*drule-tac* $y = y$ **in** *nothing-in-emptyset*)+, *blast*)
apply (*rule asynfsm.unfold*)
apply (*clarsimp simp*: *step-def steps-access actions-access*)
apply (*unfold asynfsm-def*, *clarsimp*)
apply (*erule-tac* $x = (q_1, In, Out, q_2)$ **in** *ballE-in*, *assumption*, *clarsimp*)
apply (*rule actsig.unfold*)
apply (*subst input-access*, *assumption*)
apply (*subst inner-access*, *assumption*)
apply (*unfold actsig-def*)
by (*clarsimp*, *rule equals0I*, (*drule-tac* $y = y$ **in** *nothing-in-emptyset*)+, *blast*)

definition *serial* :: $('q,'act)asynfsm \Rightarrow bool$
where $serial\ A \equiv \forall q.\ \forall In.\ \forall Out.\ \forall q'.\ step\ A\ q\ In\ Out\ q' \longrightarrow (\exists x.\ In = \{x\})$
definition *deterministic* :: $('q,'act)asynfsm \Rightarrow bool$
where $deterministic\ A \equiv \forall q.\ \forall In.\ \exists!Out.\ \exists!q'.\ step\ A\ q\ In\ Out\ q'$

definition *isomorphic* :: $('q_1,'act)asynfsm \Rightarrow ('q_2,'act)asynfsm \Rightarrow bool$
where $isomorphic\ A\ B \equiv actions\ A = actions\ B \land (\exists \varphi.\ \varphi\ (initial\ A) = initial\ B \land$
$(\forall q.\ \forall In.\ \forall Out.\ \forall q'.\ step\ A\ q\ In\ Out\ q' = step\ B\ (\varphi\ q)\ In\ Out\ (\varphi\ q')))$

definition *match* :: $'act\ actsig \Rightarrow 'act\ actsig \Rightarrow bool$
where $match\ \Sigma\ \Sigma' \equiv input\ \Sigma \cap input\ \Sigma' = \{\} \land$
$output\ \Sigma \cap output\ \Sigma' = \{\} \land$
$(input\ \Sigma \cup output\ \Sigma \cup inner\ \Sigma) \cap inner\ \Sigma' = \{\} \land$
$(input\ \Sigma' \cup output\ \Sigma' \cup inner\ \Sigma') \cap inner\ \Sigma = \{\}$

lemma *implfalse*: $\neg P \Longrightarrow P \longrightarrow Q$ **by** *blast*

definition *set-aggr-filter* **where** $set\text{-}aggr\text{-}filter\ F\ L \equiv foldl\ (\lambda Sum\ S.\ Sum \cup (S \cap F))\ \{\}\ L$

lemma *set-aggr-filter.lemma1*: $a \cap F = a \cap F \cup \{\} \cap F$ **by** *blast*

lemma *set-aggr-filter.absorb* [*simp*]: $set\text{-}aggr\text{-}filter\ F\ list \cap F = set\text{-}aggr\text{-}filter\ F\ list$
apply (*simp add*: *set-aggr-filter-def*)
apply (*induct-tac list*, *simp*)
apply *simp*
apply (*subgoal-tac* $foldl\ (\lambda Sum\ S.\ Sum \cup S \cap F)\ (a \cap F \cup \{\} \cap F)\ list =$
$a \cap F \cup foldl\ (\lambda Sum\ S.\ Sum \cup S \cap F)\ \{\}\ list \cap F$)
prefer *2*
apply (*rule semigroup-add.foldl-assoc*)
apply (*simp add*: *semigroup-add-def*, *blast*)
apply (*subst* (*3*) *set-aggr-filter.lemma1*)
by (*simp*, *blast*)

lemma *set-aggr-filter.univ-is-union* [*simp*]: $set\text{-}aggr\text{-}filter\ UNIV\ list = foldl\ op \cup \{\}\ list$
by (*simp add*: *set-aggr-filter-def*)

lemma *set-aggr-filter.associative* [*simp*]:
shows *set-aggr-filter F* (*a* # *list*) = (*a* ∩ *F*) ∪ *set-aggr-filter F list*
and *set-aggr-filter F* (*list* @ [*a*]) = (*a* ∩ *F*) ∪ *set-aggr-filter F list*
apply *succeed*
  apply (*simp add*: *set-aggr-filter-def*)
  apply (*subgoal-tac foldl* (λ*Sum S*. *Sum* ∪ *S* ∩ *F*) (*a* ∩ *F* ∪ {} ∩ *F*) *list* =
      a ∩ *F* ∪ *foldl* (λ*Sum S*. *Sum* ∪ *S* ∩ *F*) {} *list* ∩ *F*)
    prefer *2*
    apply (*rule semigroup-add.foldl-assoc*)
    apply (*simp add*: *semigroup-add-def*, *blast*)
  apply (*subst* (*2*) *set-aggr-filter.lemma1*)
  apply (*erule trans*)
  apply (*rule-tac f* = λ*x*. *a* ∩ *F* ∪ *x* **in** *eq-cong-fun-app*)
  apply (*fold set-aggr-filter-def*)
  apply (*rule set-aggr-filter.absorb*)
by (*simp add*: *set-aggr-filter-def*, *blast*)

lemma *set-aggr-filter.rev* [*simp*]: *set-aggr-filter F* (*rev list*) = *set-aggr-filter F list*
by (*induct-tac list*, *simp-all add*: *set-aggr-filter.associative*)

lemma *set-aggr-filter.zero* [*simp*]: *set-aggr-filter F* [] = {}
by (*simp add*: *set-aggr-filter-def*)

lemma *set-aggr-filter.addsub* [*rule-format*]:
  set-aggr-filter Add List ⊆ *P* ⟶ *set-aggr-filter* (*P* − *M*) *List* ∩ *Sub* = {} ⟶
  set-aggr-filter (*P* − *M*) *List* = *set-aggr-filter* ((*Add* ∪ *P*) − (*Sub* ∪ *M*)) *List*
apply (*induct-tac List*, *simp*)
apply (*rule impI*)
apply *clarsimp*
apply (*subgoal-tac a* ∩ (*P* − *M*) = *a* ∩ ((*Add* ∪ *P*) − (*M* ∪ *Sub*)))
  apply *blast*
by *blast*

lemma *set-aggr-filter.subset-of-filter* [*rule-format*]: *set-aggr-filter F List* ⊆ *F*
apply (*induct-tac List*, *simp*)
by *bestsimp*

definition *bool-and-map* **where** *bool-and-map f L* ≡ *foldl* (λ*t e*. *t* ∧ *f e*) *True L*

lemma *bool-and-map.associative* [*simp*]:
shows *bool-and-map f* (*a* # *List*) = (*f a* ∧ *bool-and-map f List*)
and *bool-and-map f* (*List* @ [*a*]) = (*f a* ∧ *bool-and-map f List*)
apply *succeed*
  apply (*simp add*: *bool-and-map-def*)
  apply (*subgoal-tac foldl* (λ*t e*. *t* ∧ *f e*) (*True* ∧ *f a*) *List* =
      (*f a* ∧ *foldl* (λ*t e*. *t* ∧ *f e*) *True List*))

  prefer *2*
  apply (*induct-tac List*, *simp*)
  apply *clarsimp*
  apply (*case-tac f aa*)
   apply *clarsimp*
  apply *clarsimp*
  apply (*subgoal-tac foldl* ($\lambda t\ e.\ t \wedge f\ e$) *False list* = *False*)
   apply *blast*
  apply (*induct-tac list*, *simp*)
  apply *simp*
 apply *bestsimp*
by (*simp add*: *bool-and-map-def*, *blast*)

lemma *bool-and-map.absorb* [*simp*]: *bool-and-map f* [] = *True*
by (*bestsimp simp*: *bool-and-map-def*)

lemma *bool-and-map.every* [*intro*,*rule-format*]: *bool-and-map f List* $\longrightarrow$ ($\forall x \in$ *set List*. *f x*)
apply (*induct-tac List*, *simp add*: *bool-and-map-def*)
by *simp*

lemma *bool-and-map.everyA*: *bool-and-map f List* $\Longrightarrow \forall x \in$ *set List*. *f x*
apply (*rule-tac P* = *bool-and-map f List* **and** *Q* = $\forall x \in$ *set List*. *f x* **in** *impE*)
by (*blast intro*: *bool-and-map.every*)+

lemma *bool-and-map.everyR* [*intro*]: $\forall x \in$ *set List*. *f x* $\Longrightarrow$ *bool-and-map f List*
apply (*rule-tac Q* = *bool-and-map f List* **and** *P* = $\forall x \in$ *set List*. *f x* **in** *impE*)
  apply (*induct-tac List*, *simp add*: *bool-and-map-def*)
by *bestsimp*+

primrec *matchFSMList* :: (('*q*,'*act*)*asynfsm*)*list* $\Rightarrow$ *bool* **where**
matchFSMList [] = *True* |
matchFSMList (*A* # *L*) = (*bool-and-map* (λe. *match* (*actions A*) (*actions e*)) *L* $\wedge$
                matchFSMList L)

definition *asynCompositionRaw* ::
(('*q*,'*act*)*asynfsm*)*list* $\Rightarrow$
 (('*act actsig*) $\times$ ('*q list* $\times$ '*act multiset*)*set* $\times$ ('*q list* $\times$ '*act multiset*) $\times$
  (('*q list* $\times$ '*act multiset*) $\times$ '*act set* $\times$ '*act set* $\times$ '*q list* $\times$ '*act multiset*)*set*)
where *asynCompositionRaw L* $\equiv$
 let inputs = (($\bigcup A \in$ *set L*. *input* (*actions A*)) $-$ ($\bigcup A \in$ *set L*. *output* (*actions A*))) *in*
 let outputs = (($\bigcup A \in$ *set L*. *output* (*actions A*)) $-$ ($\bigcup A \in$ *set L*. *input* (*actions A*))) *in*
 let inners = (($\bigcup A \in$ *set L*. *inner* (*actions A*)) $\cup$ ($\bigcup A \in$ *set L*. *input* (*actions A*) $\cap$
  ($\bigcup A \in$ *set L*. *output* (*actions A*)))) *in*
 let Q = ((*list-times-compr L* (λA. *states A*)) $\times$ (*powermultiset inners*)) *in*
 (
  (*Abs-actsig* (*inputs*, *outputs*, *inners*)),
  Q,

$((map\ (\lambda A.\ initial\ A)\ L),\ \{\#\}),$
$\{(q_1,\ In,\ Out,\ q_2).\ \exists\, Inl.\ \exists\, Outl.\ let\ (ql_1,\ msg_1) = q_1\ in\ let\ (ql_2,\ msg_2) = q_2\ in$
$bool\text{-}and\text{-}map\ (\lambda(qi_1,\ ini,\ outi,\ qi_2,\ Ai).$
$((step\ Ai\ qi_1\ ini\ outi\ qi_2 \wedge multiset\text{-}of\ (ini \cap input\ (actions\ Ai) \cap inners) \subseteq\#\ msg_1)$
$\vee\ (ini = \{\} \wedge outi = \{\} \wedge qi_1 = qi_2)))$
$(zip\ ql_1\ (zip\ Inl\ (zip\ Outl\ (zip\ ql_2\ L)))) \wedge$
$In = set\text{-}aggr\text{-}filter\ (inputs \cup inners)\ Inl \wedge In \neq \{\} \wedge$
$Out = set\text{-}aggr\text{-}filter\ outputs\ Outl \wedge$
$msg_2 = (msg_1 - multiset\text{-}of\ In) + multiset\text{-}of\ (set\text{-}aggr\text{-}filter\ inners\ Outl) \wedge$
$q_1 \in Q \wedge q_2 \in Q \wedge$
$length\ ql_1 = length\ L \wedge length\ Inl = length\ L \wedge length\ Outl = length\ L \wedge$
$length\ ql_2 = length\ L\}$
)

definition $asynComposition :: (('q,'act)asynfsm)list \Rightarrow ('q\ list \times 'act\ multiset,'act)asynfsm$
where $asynComposition\ L \equiv Abs\text{-}asynfsm\ (asynCompositionRaw\ L)$

lemma $matchFSMList\text{-}no\text{-}conflict\text{-}front$ $[intro]$:
shows $matchFSMList\ (A\ \#\ list) \Longrightarrow$
$(input\ (actions\ A) \cap (\bigcup A \in set\ list.\ input\ (actions\ A))) = \{\}$
and $matchFSMList\ (A\ \#\ list) \Longrightarrow$
$(output\ (actions\ A) \cap (\bigcup A \in set\ list.\ output\ (actions\ A))) = \{\}$
and $matchFSMList\ (A\ \#\ list) \Longrightarrow$
$(inner\ (actions\ A) \cap (\bigcup A \in set\ list.\ input\ (actions\ A) \cup output\ (actions\ A) \cup$
$inner\ (actions\ A))) = \{\}$
and $matchFSMList\ (A\ \#\ list) \Longrightarrow$
$((input\ (actions\ A) \cup output\ (actions\ A) \cup inner\ (actions\ A)) \cap$
$(\bigcup A \in set\ list.\ inner\ (actions\ A))) = \{\}$
apply *succeed*
apply (*clarsimp simp*: *matchFSMList-def*)
apply (*rule equals0I*)
apply *clarsimp*
apply (*drule-tac List* = *list* **and** $f = \lambda e.\ match\ (actions\ A)\ (actions\ e)$ **and** $x = Aa$
in *bool-and-map.every*, *assumption*)
apply (*clarsimp simp*: *match-def*)
apply *blast*
apply (*clarsimp simp*: *matchFSMList-def*)
apply (*rule equals0I*)
apply *clarsimp*
apply (*drule-tac List* = *list* **and** $f = \lambda e.\ match\ (actions\ A)\ (actions\ e)$ **and** $x = Aa$
in *bool-and-map.every*, *assumption*)
apply (*clarsimp simp*: *match-def*)
apply *blast*
apply (*clarsimp simp*: *matchFSMList-def*)
apply (*rule equals0I*)
apply *clarsimp*
apply (*drule-tac List* = *list* **and** $f = \lambda e.\ match\ (actions\ A)\ (actions\ e)$ **and** $x = Aa$

in *bool-and-map.every*, *assumption*)
apply (*clarsimp simp*: *match-def*)
apply *blast*
apply (*clarsimp simp*: *matchFSMList-def*)
apply (*rule equals0I*)
apply *clarsimp*
apply (*drule-tac List* = *list* **and** $f = \lambda e.$ *match* (*actions A*) (*actions e*) **and** $x = Aa$
in *bool-and-map.every*, *assumption*)
apply (*clarsimp simp*: *match-def*)
by *blast*

lemma *Union-Bun-distrib*: $(\bigcup a \in A.\ S\ a \cup T\ a) = (\bigcup a \in A.\ S\ a) \cup (\bigcup a \in A.\ T\ a)$ **by** *blast*

lemma *abstraction*: $[\![\bigwedge x.\ P\ x]\!] \Longrightarrow P\ x$
apply (*erule-tac* $x = x$ **in** *meta-allE*)
by *assumption*

lemma *meta-abstraction*: $[\![Q\ x;\ \bigwedge x.\ Q\ x \Longrightarrow P\ x]\!] \Longrightarrow P\ x$
apply (*erule-tac* $x = x$ **in** *meta-allE*)
by *blast*

lemma *meta-abstraction6*:
$[\![Q\ a\ b\ c\ d\ e\ f;\ \bigwedge a\ b\ c\ d\ e\ f.\ Q\ a\ b\ c\ d\ e\ f \Longrightarrow P\ a\ b\ c\ d\ e\ f]\!] \Longrightarrow P\ a\ b\ c\ d\ e\ f$
apply (*erule-tac* $x = a$ **in** *meta-allE*)
apply (*erule-tac* $x = b$ **in** *meta-allE*)
apply (*erule-tac* $x = c$ **in** *meta-allE*)
apply (*erule-tac* $x = d$ **in** *meta-allE*)
apply (*erule-tac* $x = e$ **in** *meta-allE*)
apply (*erule-tac* $x = f$ **in** *meta-allE*)
by *blast*

lemma *matchFSMList-produces-actsig-lemma2*: $[\![x \in A;\ x \in B;\ A \cap B = \{\}]\!] \Longrightarrow$ *False*
by *blast*
lemma *matchFSMList-produces-actsig-lemma1*:
$[\![(d - e) \cap (e - d) = \{\};\ (d - e) \cap (f \cup d \cap e) = \{\} \wedge (e - d) \cap (f \cup d \cap e) = \{\};$
$a \cap d = \{\};\ b \cap e = \{\} \wedge c \cap (d \cup e \cup f) = \{\};\ \ (a \cup b \cup c) \cap f = \{\};$
$a \cap b = \{\};\ a \cap c = \{\};\ b \cap c = \{\}]\!] \Longrightarrow$
$(a \cup d - (b \cup e)) \cap (b \cup e - (a \cup d)) =$
$\{\} \wedge (a \cup d - (b \cup e)) \cap (c \cup f \cup (a \cup d) \cap (b \cup e)) = \{\} \wedge$
$(b \cup e - (a \cup d)) \cap (c \cup f \cup (a \cup d) \cap (b \cup e)) = \{\}$
by ((*rule conjI*)?, *rule equals0I*, ((*drule-tac* $y = y$ **in** *nothing-in-emptyset*)+, *blast*))+

lemma *matchFSMList-produces-actsig* [*rule-format*]:
matchFSMList L $\longrightarrow$ (
$(\bigcup A{\in}set\ L.\ input\ (actions\ A)) - (\bigcup A{\in}set\ L.\ output\ (actions\ A)),$
$(\bigcup A{\in}set\ L.\ output\ (actions\ A)) - (\bigcup A{\in}set\ L.\ input\ (actions\ A)),$
$(\bigcup A{\in}set\ L.\ inner\ (actions\ A)) \cup (\bigcup A{\in}set\ L.\ input\ (actions\ A)) \cap$

$\quad(\bigcup A{\in}set\ L.\ output\ (actions\ A))$
$\ ) \in actsig$
apply (*induct-tac L, simp add: actsig-def*)
apply (*rule impI*)
apply (*insert matchFSMList-no-conflict-front*)
apply (*erule-tac x = a* **in** *meta-allE*)+
apply (*erule-tac x = list* **in** *meta-allE*)+
apply (*simp add: actsig-def*)

apply (*insert Union-Bun-distrib*)[*1*]
apply (*erule-tac x = set list* **in** *meta-allE*)
apply (*erule-tac* $x = \lambda a.\ (input\ (actions\ a) \cup output\ (actions\ a))$ **in** *meta-allE*)
apply (*erule-tac* $x = \lambda a.\ inner\ (actions\ a)$ **in** *meta-allE*)
apply *simp*

apply (*insert Union-Bun-distrib*)[*1*]
apply (*erule-tac x = set list* **in** *meta-allE*)
apply (*erule-tac* $x = \lambda a.\ input\ (actions\ a)$ **in** *meta-allE*)
apply (*erule-tac* $x = \lambda a.\ output\ (actions\ a)$ **in** *meta-allE*)
apply *simp*

apply (*subgoal-tac* $input\ (actions\ a) \cap output\ (actions\ a) = \{\} \wedge$
$\quad input\ (actions\ a) \cap inner\ (actions\ a) = \{\} \wedge$
$\quad output\ (actions\ a) \cap inner\ (actions\ a) = \{\}$)
 prefer *2*
 apply (*rule actsig.unfold*)
 apply (*force simp: actsig-def*)

apply (*rule-tac* $a = input\ (actions\ a)$ **and**
$\qquad b = output\ (actions\ a)$ **and**
$\qquad c = inner\ (actions\ a)$ **and**
$\qquad d = \bigcup a{\in}set\ list.\ input\ (actions\ a)$ **and**
$\qquad e = \bigcup a{\in}set\ list.\ output\ (actions\ a)$ **and**
$\qquad f = \bigcup a{\in}set\ list.\ inner\ (actions\ a)$ **and**
$\qquad Q = \lambda a\ b\ c\ d\ e\ f.\ (d - e) \cap (e - d) = \{\} \wedge$
$\qquad\quad (d - e) \cap (f \cup d \cap e) = \{\} \wedge (e - d) \cap (f \cup d \cap e) = \{\} \wedge$
$\qquad\quad a \cap d = \{\} \wedge b \cap e = \{\} \wedge c \cap (d \cup e \cup f) = \{\} \wedge$
$\qquad\quad (a \cup b \cup c) \cap f = \{\} \wedge$
$\qquad\quad a \cap b = \{\} \wedge a \cap c = \{\} \wedge b \cap c = \{\}$ **in** *meta-abstraction6*)
 apply *simp*
apply (*rename-tac Ai Ao Ainner Li Lo Linner*)

by (*rule-tac a = Ai* **and** *b = Ao* **and** *c = Ainner* **and** *d = Li* **and** *e = Lo* **and** *f = Linner* **in** *matchFSMList-produces-actsig-lemma1, simp+*)

lemma *asynCompositionValid* [*intro*]: *matchFSMList L* $\Longrightarrow$ *asynCompositionRaw L* $\in$ *asynfsm*
apply (*simp add: asynCompositionRaw-def asynfsm-def Let-def*)

apply (*rule conjI*)
  apply (*simp add*: *list-times-compr-def*)
  apply (*induct-tac L*, *simp*)
  apply (*simp add*: *initial-in-states*)
apply (*rule conjI*)
  apply (*simp add*: *powermultiset-def*)
apply (*rule allI*)+
apply (*rename-tac* ql_1 M_1 *In Out* ql_2 M_2)
apply (*rule impI*)
apply ((*erule exE*)+, (*erule conjE*)+)
apply ((*rule conjI*)?, *assumption*)+
apply (*rule conjI*)
  apply (*subst output-access*, *fastsimp simp*: *matchFSMList-produces-actsig*)
  apply (*clarsimp simp*: *set-aggr-filter.subset-of-filter*)
apply (*subst input-access*, *fastsimp simp*: *matchFSMList-produces-actsig*)
apply (*subst inner-access*, *fastsimp simp*: *matchFSMList-produces-actsig*)
by (*clarsimp simp*: *set-aggr-filter.subset-of-filter*)

lemma *asynCompositionValidSubst* [*simp*]:
matchFSMList L $\Longrightarrow$
  Rep-asynfsm (*Abs-asynfsm* (*asynCompositionRaw L*)) = *asynCompositionRaw L*
by (*bestsimp simp*: *Abs-asynfsm-inverse asynCompositionValid*)

lemma *asynCompositionCommutative*:
⟦*matchFSMList* [*A*, *B*]; *matchFSMList* [*B*, *A*]⟧ $\Longrightarrow$
  isomorphic (*asynComposition* [*A*, *B*]) (*asynComposition* [*B*, *A*])
apply (*insert asynCompositionValid*[**where** *L* = [*A*, *B*]])
apply (*insert asynCompositionValid*[**where** *L* = [*B*, *A*]])
apply *simp*

apply (*unfold isomorphic-def*)
apply (*rule conjI*)
  apply (*simp add*: *asynComposition-def asynCompositionRaw-def Let-def*)
  apply ((*unfold match-def*)[*1*], (*erule conjE*)+)
  apply (*subgoal-tac*
    (*input* (*actions A*) ∪ *input* (*actions B*) − (*output* (*actions A*) ∪ *output* (*actions B*))) =
      (*input* (*actions B*) ∪ *input* (*actions A*) − (*output* (*actions B*) ∪ *output* (*actions A*))) ∧
    (*output* (*actions A*) ∪ *output* (*actions B*) − (*input* (*actions A*) ∪ *input* (*actions B*))) =
      (*output* (*actions B*) ∪ *output* (*actions A*) − (*input* (*actions B*) ∪ *input* (*actions A*))) ∧
    (*inner* (*actions A*) ∪ *inner* (*actions B*) ∪ (*input* (*actions A*) ∪ *input* (*actions B*)) ∩
        (*output* (*actions A*) ∪ *output* (*actions B*))) =
      (*inner* (*actions B*) ∪ *inner* (*actions A*) ∪ (*input* (*actions B*) ∪ *input* (*actions A*)) ∩
        (*output* (*actions B*) ∪ *output* (*actions A*))))
    apply ((*erule conjE*)+, *simp*)
  apply *blast*

The next line gives the actual mapping function.

apply (*rule-tac* $x = \lambda(L, M).\ (rev\ L, M)$ **in** *exI*)
apply (*simp add*: *split-def*)
apply (*rule conjI*)
  apply (*simp add*: *asynComposition-def asynCompositionRaw-def Let-def*)
apply (*rule allI*)+
apply (*rename-tac* $ql_1\ M_1$ *In Out* $ql_2\ M_2$)
apply (*simp add*: *asynComposition-def asynCompositionRaw-def Let-def step-def*)
apply (*rule iffI*)
  apply (*erule exE*)+
  apply (*rule-tac* $x = rev\ Inl$ **in** *exI*)
  apply (*rule-tac* $x = rev\ Outl$ **in** *exI*)

The following organizes meaningful names for the components of the composite lists and proves that they indeed have length two.

  apply (*subgoal-tac* $(\exists\, qla_1\ qlb_1.\ ql_1 = [qla_1, qlb_1]) \wedge (\exists\, Inla\ Inlb.\ Inl = [Inla, Inlb]) \wedge$
  $(\exists\, Outla\ Outlb.\ Outl = [Outla, Outlb]) \wedge (\exists\, qla_2\ qlb_2.\ ql_2 = [qla_2, qlb_2])$)
    prefer *2*
    apply (*subgoal-tac* $ql_1 = [hd\ ql_1, hd\ (tl\ ql_1)] \wedge Inl = [hd\ Inl, hd\ (tl\ Inl)] \wedge$
    $Outl = [hd\ Outl, hd\ (tl\ Outl)] \wedge ql_2 = [hd\ ql_2, hd\ (tl\ ql_2)]$)
      prefer *2* **apply** ((*erule conjE*)+, ((*rule conjI*)?, *simp add*: *list-fixlen-expl2*)+)[*1*]
    apply (*erule conjE*)+
    apply (*rule conjI*, *rule-tac* $x = hd\ ql_1$ **in** *exI*, *rule-tac* $x = hd\ (tl\ ql_1)$ **in** *exI*, *assumption*)
    apply (*rule conjI*, *rule-tac* $x = hd\ Inl$ **in** *exI*, *rule-tac* $x = hd\ (tl\ Inl)$ **in** *exI*, *assumption*)
    apply (*rule conjI*, *rule-tac* $x = hd\ Outl$ **in** *exI*, *rule-tac* $x = hd\ (tl\ Outl)$ **in** *exI*, *assumption*)
    apply (*rule-tac* $x = hd\ ql_2$ **in** *exI*, *rule-tac* $x = hd\ (tl\ ql_2)$ **in** *exI*, *assumption*)
  apply (*erule conjE*)+
  apply (*erule exE*)+

Main proof line continues below.

  apply (*rule conjI*, *simp*)
    apply (*rule conjI*)
      apply (*case-tac* $Inlb = \{\}$, *force*)
      apply (*subst Un-commute*[*of inner* (*actions B*) *inner* (*actions A*)])
      apply (*subst Un-commute*[*of input* (*actions B*) *input* (*actions A*)])
      apply (*subst Un-commute*[*of output* (*actions B*) *output* (*actions A*)])
      apply *simp*
    apply (*case-tac* $Inla = \{\}$, *force*)
    apply (*subst Un-commute*[*of inner* (*actions B*) *inner* (*actions A*)])
    apply (*subst Un-commute*[*of input* (*actions B*) *input* (*actions A*)])
    apply (*subst Un-commute*[*of output* (*actions B*) *output* (*actions A*)])
    apply *simp*
  apply (*rule conjI*, *simp*, *blast*)
  apply (*rule conjI*, *assumption*)
  apply (*rule conjI*, *simp*, *blast*)
  apply (*rule conjI*)
    apply *simp*

apply (*rule-tac f* = *DrahflowTools.multiset-of* **in** *eq-cong-fun-app*)
apply *blast*

apply (*subgoal-tac*
(*inner* (*actions B*) ∪ *inner* (*actions A*) ∪ (*input* (*actions B*) ∪ *input* (*actions A*)) ∩
(*output* (*actions B*) ∪ *output* (*actions A*))) =
(*inner* (*actions A*) ∪ *inner* (*actions B*) ∪ (*input* (*actions A*) ∪ *input* (*actions B*)) ∩
(*output* (*actions A*) ∪ *output* (*actions B*))))
prefer *2* **apply** *blast*
apply ((*rule conjI*)*?*, *simp*)+

apply (*erule exE*)+
apply (*rule-tac x* = *rev Inl* **in** *exI*)
apply (*rule-tac x* = *rev Outl* **in** *exI*)

The following organizes meaningful names for the components of the composite lists and proves that they indeed have length two.

apply (*subgoal-tac* ($\exists$ qla_1 qlb_1. ql_1 = [qla_1, qlb_1]) ∧ ($\exists$ *Inla Inlb*. *Inl* = [*Inla*, *Inlb*]) ∧
($\exists$ *Outla Outlb*. *Outl* = [*Outla*, *Outlb*]) ∧ ($\exists$ qla_2 qlb_2. ql_2 = [qla_2, qlb_2]))
prefer *2*
apply (*subgoal-tac* ql_1 = [*hd* ql_1, *hd* (*tl* ql_1)] ∧ *Inl* = [*hd Inl*, *hd* (*tl Inl*)] ∧
Outl = [*hd Outl*, *hd* (*tl Outl*)] ∧ ql_2 = [*hd* ql_2, *hd* (*tl* ql_2)])
prefer *2* **apply** ((*erule conjE*)+, ((*rule conjI*)*?*, *simp add*: *list-fixlen-expl2*)+)[*1*]
apply (*erule conjE*)+
apply (*rule conjI*, *rule-tac x* = *hd* ql_1 **in** *exI*, *rule-tac x* = *hd* (*tl* ql_1) **in** *exI*, *assumption*)
apply (*rule conjI*, *rule-tac x* = *hd Inl* **in** *exI*, *rule-tac x* = *hd* (*tl Inl*) **in** *exI*, *assumption*)
apply (*rule conjI*, *rule-tac x* = *hd Outl* **in** *exI*, *rule-tac x* = *hd* (*tl Outl*) **in** *exI*, *assumption*)
apply (*rule-tac x* = *hd* ql_2 **in** *exI*, *rule-tac x* = *hd* (*tl* ql_2) **in** *exI*, *assumption*)
apply (*erule conjE*)+
apply (*erule exE*)+

Main proof line continues below.

apply (*rule conjI*)
apply *simp*
apply (*rule conjI*)
apply (*case-tac Inlb* = {}, *force*)
apply (*subst Un-commute*[*of inner* (*actions A*) *inner* (*actions B*)])
apply (*subst Un-commute*[*of input* (*actions A*) *input* (*actions B*)])
apply (*subst Un-commute*[*of output* (*actions A*) *output* (*actions B*)])
apply *simp*
apply (*case-tac Inla* = {}, *force*)
apply (*subst Un-commute*[*of inner* (*actions A*) *inner* (*actions B*)])
apply (*subst Un-commute*[*of input* (*actions A*) *input* (*actions B*)])
apply (*subst Un-commute*[*of output* (*actions A*) *output* (*actions B*)])
apply *simp*

apply (*rule conjI*, *simp*, *blast*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *simp*, *blast*)

```
apply (rule conjI)
  apply simp
  apply (rule-tac f = DrahflowTools.multiset-of in eq-cong-fun-app)
  apply blast
apply (subgoal-tac
   (inner (actions B) ∪ inner (actions A) ∪ (input (actions B) ∪ input (actions A)) ∩
     (output (actions B) ∪ output (actions A))) =
   (inner (actions A) ∪ inner (actions B) ∪ (input (actions A) ∪ input (actions B)) ∩
     (output (actions A) ∪ output (actions B))))
  prefer 2 apply blast
by ((rule conjI)?, simp)+

lemma matchFSMList.trivial [intro]: matchFSMList [] by simp
lemma matchFSMList.inherit [rule-format]: matchFSMList (A # L) ⟶ matchFSMList L
by bestsimp

lemma set-aggr-filter-assoc-finite [intro,rule-format]:
(∀ x ∈ set List. finite x) ⟶ finite (set-aggr-filter F List)
apply (induct-tac List)
  apply (bestsimp simp: set-aggr-filter-def)
by simp

lemma step-not-empty [intro]: ¬ step A q1 {} Out q2
by (force simp: step-def asynfsm-def intro: asynfsm.unfold)

lemma list-times-compr-same-length [simp,rule-format]:
∀ x. x ∈ list-times-compr L f ⟶ length x = length L
by (induct-tac L, simp+)

lemma step-asyn-implies-length [simp]:
shows ⟦matchFSMList L; step (asynComposition L) q1 In Out q2⟧ ⟹
  length (fst q1) = length L
and ⟦matchFSMList L; step (asynComposition L) q1 In Out q2⟧ ⟹ length (fst q2) = length L
  apply (simp add: step-def)
  apply (rule-tac A = (asynComposition L) in asynfsm.unfold)
  apply (simp add: asynfsm-def)
  apply (erule-tac conjE)
  apply (erule-tac x = (q1, In, Out, q2) in ballE-in, assumption)
  apply clarsimp
  apply (thin-tac Out ⊆ output (actions (asynComposition L)))
  apply (thin-tac In ⊆ input (actions (asynComposition L)) ∪
    inner (actions (asynComposition L)))
  apply (thin-tac In ≠ {})
  apply (thin-tac initial (asynComposition L) ∈ states (asynComposition L))
  apply (thin-tac (q1, In, Out, q2) ∈ steps (asynComposition L))
  apply (thin-tac q2 ∈ states (asynComposition L))
  apply (drule-tac asynCompositionValid)
```

apply (*bestsimp simp*: *asynComposition-def asynCompositionRaw-def Let-def*)
apply (*simp add*: *step-def*)
apply (*rule-tac* $A = (asynComposition\ L)$ **in** *asynfsm.unfold*)
apply (*simp add*: *asynfsm-def*)
apply (*erule-tac conjE*)
apply (*erule-tac* $x = (q_1,\ In,\ Out,\ q_2)$ **in** *ballE-in*, *assumption*)
apply *clarsimp*
apply (*thin-tac* $Out \subseteq output\ (actions\ (asynComposition\ L)))$
apply (*thin-tac* $In \subseteq input\ (actions\ (asynComposition\ L)) \cup$
$inner\ (actions\ (asynComposition\ L)))$
apply (*thin-tac* $In \neq \{\})$
apply (*thin-tac* $initial\ (asynComposition\ L) \in states\ (asynComposition\ L))$
apply (*thin-tac* $(q_1,\ In,\ Out,\ q_2) \in steps\ (asynComposition\ L))$
apply (*thin-tac* $q_1 \in states\ (asynComposition\ L))$
apply (*drule-tac asynCompositionValid*)
by (*bestsimp simp*: *asynComposition-def asynCompositionRaw-def Let-def*)

lemma *in-set-implies-index* [*intro*,*rule-format*]: $x \in set\ L \longrightarrow (\exists\, i.\ L\ !\ i = x \wedge i < length\ L)$
apply (*induct-tac L*)
apply *simp*
apply *simp*
apply (*rule conjI*)
apply (*rule impI*, *rule-tac* $x = 0$ **in** *exI*, *simp*)
apply (*rule impI*)
apply (*erule-tac impE*, *assumption*)
apply (*erule-tac exE*)
apply (*rule-tac* $x = Suc\ i$ **in** *exI*)
by *simp*

lemma *list-index-shift* [*intro*,*rule-format*]:
$\forall\, i.\ i < Suc\ (length\ list) \longrightarrow 0 < i \longrightarrow (a\ \#\ list)\ !\ i = list\ !\ (i - 1)$
apply (*induct-tac list*, *simp*)
apply (*rule allI*)
apply (*case-tac i*, *simp*)
by *simp*

lemma *matchFSMList-shared-same-index* [*intro*,*simp*,*rule-format*]:
$matchFSMList\ L \longrightarrow I \neq \{\} \longrightarrow$
$(\forall\, i.\ i < length\ L \longrightarrow (\forall\, j.\ j < length\ L \longrightarrow$
$I \subseteq input\ (actions\ (L\ !\ i)) \cup inner\ (actions\ (L\ !\ i)) \longrightarrow$
$I \subseteq input\ (actions\ (L\ !\ j)) \cup inner\ (actions\ (L\ !\ j)) \longrightarrow i = j))$
apply (*induct-tac L*, *simp*)
apply *clarsimp*
apply (*case-tac* $i = 0$)
apply (*case-tac* $j = 0$)
apply *blast*
apply (*subgoal-tac* $(a\ \#\ list)\ !\ i = a$)

```
  prefer 2 apply simp
 apply (frule-tac bool-and-map.everyA)
 apply (erule-tac x = list ! (j − 1) in ballE-in, simp)
 apply (unfold match-def)[1]
 apply (subgoal-tac (a # list) ! j = list ! (j − 1))
  prefer 2 apply (erule list-index-shift, simp)
 apply clarsimp
 apply (rule-tac a = input (actions a) and b = output (actions a) and c = inner (actions a)
  and d = input (actions (list ! (j − Suc 0))) and e = output (actions (list ! (j − Suc 0)))
  and f = inner (actions (list ! (j − Suc 0)))
  and Q = λa b c d e f. I ≠ {} ∧ I ⊆ a ∪ c ∧ I ⊆ d ∪ f ∧ (a ∪ b ∪ c) ∩ f = {} ∧
   (d ∪ e ∪ f) ∩ c = {} in meta-abstraction6)
  apply blast
 apply (erule conjE)+
 apply (subgoal-tac I = {}, simp)
 apply (rule equals0I)
 apply (drule-tac y = y in nothing-in-emptyset)+
 apply blast
apply (case-tac j = 0)
 apply (subgoal-tac (a # list) ! j = a)
  prefer 2 apply simp
 apply (frule-tac bool-and-map.everyA)
 apply (erule-tac x = list ! (i − 1) in ballE-in, simp)
 apply (unfold match-def)[1]
 apply (subgoal-tac (a # list) ! i = list ! (i − 1))
  prefer 2 apply (erule list-index-shift, simp)
 apply clarsimp
 apply (rule-tac a = input (actions a) and b = output (actions a) and c = inner (actions a)
  and d = input (actions (list ! (i − Suc 0))) and e = output (actions (list ! (i − Suc 0)))
  and f = inner (actions (list ! (i − Suc 0)))
  and Q = λa b c d e f. I ≠ {} ∧ I ⊆ a ∪ c ∧ I ⊆ d ∪ f ∧ (a ∪ b ∪ c) ∩ f = {} ∧
   (d ∪ e ∪ f) ∩ c = {} in meta-abstraction6)
  apply blast
 apply (erule conjE)+
 apply (subgoal-tac I = {}, simp)
 apply (rule equals0I)
 apply (drule-tac y = y in nothing-in-emptyset)+
 apply blast
apply (erule-tac x = i − 1 in allE)
apply (subgoal-tac i − 1 < length list)
 prefer 2 apply simp
apply (erule-tac impE, assumption)
apply (erule-tac x = j − 1 in allE)
apply (subgoal-tac j − 1 < length list)
 prefer 2 apply simp
apply (erule-tac impE, assumption)
apply (subgoal-tac (a # list) ! i = list ! (i − 1))
```

  prefer *2* **apply** (*erule list-index-shift*, *simp*)
apply (*subgoal-tac* $(a \# list) ! j = list ! (j - 1)$)
  prefer *2* **apply** (*erule list-index-shift*, *simp*)
by *simp*

lemma *disjE-excl1*: $[\![P \lor Q; [\![P; \neg Q]\!] \Longrightarrow R; Q \Longrightarrow R]\!] \Longrightarrow R$ **by** *blast*
lemma *disjE-excl2*: $[\![P \lor Q; P \Longrightarrow R; [\![\neg P; Q]\!] \Longrightarrow R]\!] \Longrightarrow R$ **by** *blast*

lemma *step-asyn-implies-finite* [*intro*]:
$[\![$*matchFSMList L*; *step* (*asynComposition L*) q_1 *In Out* q_2; $\bigwedge x.\ x \in$ *set L* $\Longrightarrow$ *serial x*$]\!]$
$\Longrightarrow$ *finite In*
apply (*frule-tac asynCompositionValid*)
apply (*clarsimp simp*: *step-def asynComposition-def asynCompositionRaw-def*
      Let-def asynCompositionValid)
apply (*rule set-aggr-filter-assoc-finite*)
apply (*rename-tac Inputi*)
apply (*case-tac Inputi* = {}, *blast*)
apply (*drule-tac bool-and-map.everyA*)
apply (*unfold serial-def*)
apply (*fold step-def*)
apply (*unfold set-zip*)
apply (*rename-tac* q_1 *M* q_2 *Inputi*)

apply (*subgoal-tac* $\exists!i.\ i <$ *length Inl* $\land$ *Inl* ! i = *Inputi* $\land$
      step (*L* ! *i*) (q_1 ! *i*) *Inputi* (*Outl* ! *i*) (q_2 ! *i*))
  prefer *2*
  apply (*rule ex-ex1I*)
    apply (*drule-tac L* = *Inl* **in** *in-set-implies-index*)
    apply (*erule exE*)
    apply (*rule-tac x* = *i* **in** *exI*)
    apply (*rule conjI*, *simp*)
    apply (*rule conjI*, *simp*)
    apply (*erule-tac x* = (q_1 ! *i*, *Inl* ! *i*, *Outl* ! *i*, q_2 ! *i*, *L* ! *i*) **in** *ballE*, *bestsimp*)
    apply (*subgoal-tac* (q_1 ! *i*, *Inl* ! *i*, *Outl* ! *i*, q_2 ! *i*, *L* ! *i*) $\in$
        {(q_1 ! *i*, *zip Inl* (*zip Outl* (*zip* q_2 *L*)) ! *i*) |
         i. *i* < *min* (*length* q_1)(*length* (*zip Inl* (*zip Outl* (*zip* q_2 *L*))))}, *blast*)
    apply (*rule CollectI*)
    apply (*rule-tac x* = *i* **in** *exI*)
    apply (*rule conjI*)
      apply (*erule-tac conjE*)+
      apply (*subst nth-zip*, *assumption*, *bestsimp*)
      apply (*subst nth-zip*)
        apply (*erule-tac t* = *length Outl* **and** *s* = *length L* **in** *ssubst*)
        apply (*erule-tac t* = *length L* **and** *s* = *length Inl* **in** *subst*)
        apply *assumption*
      apply (*subgoal-tac length* q_2 = *length L*)
        apply (*subst length-zip*)

apply (*erule-tac* t = *length* q_2 **in** *ssubst*)
apply (*subst lower-semilattice-locale.min-max.less-eq-less-inf.inf-idem*)
apply (*erule-tac* t = *length* L **and** s = *length Inl* **in** *subst*)
apply *assumption*
apply (*rule-tac* f = *states* **in** *list-times-compr-same-length*)
apply *assumption*
apply (*subst nth-zip*)
apply (*subgoal-tac length* q_2 = *length* L)
apply (*erule-tac* t = *length* q_2 **and** s = *length* L **in** *ssubst*)
apply (*erule-tac* t = *length* L **and** s = *length Inl* **in** *subst*)
apply *assumption*
apply (*rule-tac* f = *states* **in** *list-times-compr-same-length*)
apply *assumption*
apply (*erule-tac* t = *length* L **and** s = *length Inl* **in** *subst*)
apply *assumption*
apply (*rule refl*)
apply *bestsimp*
apply (*rename-tac* i j)
apply (*subgoal-tac Inputi* $\subseteq$ *input* (*actions* (L ! i)) $\cup$ *inner* (*actions* (L ! i)))
apply (*subgoal-tac Inputi* $\subseteq$ *input* (*actions* (L ! j)) $\cup$ *inner* (*actions* (L ! j)))
apply (*frule-tac* $L = L$ **and** I = *Inputi* **and** $i = i$ **and** $j = j$
in *matchFSMList-shared-same-index*)
apply (*simp*)+
apply (*erule-tac conjE*)+
apply (*erule-tac step-respects-signature*)
apply (*erule-tac conjE*)+
apply (*erule-tac step-respects-signature*)
apply (*erule ex1E*)
apply (*erule conjE*)+
apply (*erule-tac* x = L ! i **in** *meta-allE*)
apply *clarsimp*
apply (*erule-tac* x = q_1 ! i
and $P = \lambda q.\ \forall In.\ (\exists Out.\ Ex\ (step\ (L\ !\ i)\ q\ In\ Out)) \longrightarrow (\exists x.\ In = \{x\})$ **in** *allE*)
apply (*erule-tac* x = *Inl* ! i
and $P = \lambda In.\ (\exists Out.\ Ex\ (step\ (L\ !\ i)\ (q_1\ !\ i)\ In\ Out)) \longrightarrow (\exists x.\ In = \{x\})$ **in** *allE*)
apply (*subgoal-tac* $(\exists Out.\ Ex\ (step\ (L\ !\ i)\ (q_1\ !\ i)\ (Inl\ !\ i)\ Out))$)
apply *simp*
apply (*erule exE*)+
apply *simp*
apply (*rule-tac* x = *Outl* ! i **in** *exI*)
apply (*rule-tac* x = q_2 ! i **in** *exI*)
by *assumption*

lemma *set-aggr-filter.element-somewhere-in-list* [*rule-format*]:
$x \in$ *set-aggr-filter* F $L \longrightarrow (\exists i.\ x \in L\ !\ i \wedge i < length\ L)$
apply (*induct-tac* L, *simp add*: *set-aggr-filter-def*)
apply *clarsimp*

```
apply (rule conjI)
  apply (rule impI)
  apply (rule-tac x = 0 in exI)
  apply simp
apply (case-tac x ∈ set-aggr-filter F list)
  apply clarsimp
  apply (rule-tac x = Suc i in exI)
  apply simp
by simp

definition source-machine L inp ≡
  (THE i. inp ∈ input (actions (L ! i)) ∪ inner (actions (L ! i)) ∧ i < length L)

lemma source-machine-input [intro]:
⟦matchFSMList L; ∀ A ∈ set L. serial A; step (asynComposition L) q1 In Out q2; inp ∈ In⟧
⟹ inp ∈ input (actions (L ! source-machine L inp)) ∪
       inner (actions (L ! source-machine L inp))
apply (frule-tac asynCompositionValid)
apply (clarsimp simp: step-def asynComposition-def asynCompositionRaw-def
           Let-def asynCompositionValid)
apply (drule-tac bool-and-map.everyA)
apply (unfold set-zip)
apply (rename-tac q1 M q2)
apply (unfold source-machine-def)

apply (subgoal-tac (λP. (λi. P i) (THE i. P i))
           (λi. inp ∈ input (actions (L ! i)) ∪ inner (actions (L ! i)) ∧ i < length L), blast)
apply (rule theI')
apply (rule ex-ex1I)
  apply (drule-tac L = Inl in set-aggr-filter.element-somewhere-in-list)
  apply (erule exE)
  apply (rule-tac x = i in exI)
  apply (erule-tac x = (q1 ! i, Inl ! i, Outl ! i, q2 ! i, L ! i) in ballE)
    apply clarsimp
    apply (erule disjE)
      apply (erule conjE)
      apply (fold step-def)
      apply (drule-tac step-respects-signature(2))
      apply blast
    apply blast
  apply (subgoal-tac (q1 ! i, Inl ! i, Outl ! i, q2 ! i, L ! i) ∈
              {(q1 ! i, zip Inl (zip Outl (zip q2 L)) ! i) |
                i. i < min (length q1) (length (zip Inl (zip Outl (zip q2 L))))}, blast)
  apply (rule CollectI)
  apply (rule-tac x = i in exI)
  apply (rule conjI)
    apply (erule-tac conjE)+
```

apply (*subst nth-zip*, *assumption*, *bestsimp*)
apply (*subst nth-zip*)
apply (*erule-tac* t = *length Outl* **and** s = *length L* **in** *ssubst*)
apply (*erule-tac* t = *length L* **and** s = *length Inl* **in** *subst*)
apply *assumption*
apply (*subgoal-tac length* q_2 = *length L*)
apply (*subst length-zip*)
apply (*erule-tac* t = *length* q_2 **in** *ssubst*)
apply (*subst lower-semilattice-locale.min-max.less-eq-less-inf.inf-idem*)
apply (*erule-tac* t = *length L* **and** s = *length Inl* **in** *subst*)
apply *assumption*
apply (*rule-tac* f = *states* **in** *list-times-compr-same-length*)
apply *assumption*
apply (*subst nth-zip*)
apply (*subgoal-tac length* q_2 = *length L*)
apply (*erule-tac* t = *length* q_2 **and** s = *length L* **in** *ssubst*)
apply (*erule-tac* t = *length L* **and** s = *length Inl* **in** *subst*)
apply *assumption*
apply (*rule-tac* f = *states* **in** *list-times-compr-same-length*)
apply *assumption*
apply (*erule-tac* t = *length L* **and** s = *length Inl* **in** *subst*)
apply *assumption*
apply (*rule refl*)
apply *bestsimp*
apply (*rename-tac i j*)
apply (*subgoal-tac* $\{inp\} \subseteq$ *input* (*actions* (L ! i)) $\cup$ *inner* (*actions* (L ! i)))
apply (*subgoal-tac* $\{inp\} \subseteq$ *input* (*actions* (L ! j)) $\cup$ *inner* (*actions* (L ! j)))
apply (*frule-tac* $L = L$ **and** $I = \{inp\}$ **and** $i = i$ **and** $j = j$
in *matchFSMList-shared-same-index*)
by (*simp*)+

lemma *source-machine-length* [*intro*]:
$[\![$*matchFSMList L*; $\forall A \in$ *set L*. *serial A*; *step* (*asynComposition L*) q_1 *In Out* q_2; *inp* $\in$ *In*$]\!]$
$\Longrightarrow$ *source-machine L inp* $<$ *length L*
apply (*simp add*: *source-machine-def*)
apply (*subgoal-tac*
(λi. (*inp* $\in$ *input* (*actions* (L ! i)) $\vee$ *inp* $\in$ *inner* (*actions* (L ! i))) $\wedge$ $i <$ *length L*)
(*THE* i. (*inp* $\in$ *input* (*actions* (L ! i)) $\vee$ *inp* $\in$ *inner* (*actions* (L ! i))) $\wedge$ $i <$ *length L*),
force)
apply (*rule theI'*)
apply (*rule ex-ex1I*)
apply (*frule-tac asynCompositionValid*)
apply (*clarsimp simp*: *step-def asynComposition-def asynCompositionRaw-def*
Let-def asynCompositionValid)
apply (*drule-tac set-aggr-filter.element-somewhere-in-list*)
apply (*erule-tac exE*)
apply (*rule-tac* $x = i$ **in** *exI*)

apply *simp*
apply (*drule-tac bool-and-map.everyA*)
apply (*rename-tac* ql_1 *y* ql_2 *i*)
apply (*subgoal-tac* (ql_1 ! *i*, *Inl* ! *i*, *Outl* ! *i*, ql_2 ! *i*, *L* ! *i*) ∈
set (*zip* ql_1 (*zip Inl* (*zip Outl* (*zip* ql_2 *L*)))))
apply (*erule-tac x* = (ql_1 ! *i*, *Inl* ! *i*, *Outl* ! *i*, ql_2 ! *i*, *L* ! *i*) **in** *ballE-in*)
apply (*simp add*: *split-def*)
apply (*case-tac Inl* ! *i* = {}, *force*)
apply *simp*
apply (*erule conjE*)+
apply (*fold step-def*)
apply (*drule-tac step-respects-signature*(*2*))
apply *force*
apply (*subst in-set-conv-nth*)
apply (*rule-tac x* = *i* **in** *exI*)
apply *force*
by (*rule-tac L* = *L* **and** *I* = {*inp*} **and** *i* = *x* **and** *j* = *y*
in *matchFSMList-shared-same-index*, *force*+)

lemma *set-aggr-filter.empty-replicate* [*simp*]: *set-aggr-filter F* (*replicate len* {}) = {}
by (*induct len*, *simp*+)

lemma *set-aggr-filter.gobble-empty-replicate* [*simp*,*rule-format*]:
i < *len* ⟶ *set-aggr-filter F* (*replicate len* {}[*i* := *L*]) = *set-aggr-filter F* [*L*]

Aggregating arbitrary amounts of empty sets does not make any difference.

by (*rule proofHole*[*of ?thesis*])

lemma *composite-actions.fold*:
assumes *matchFSMList L*
shows (⋃*A* ∈ *set L*. *input* (*actions A*)) − (⋃*A* ∈ *set L*. *output* (*actions A*)) =
input (*actions* (*asynComposition L*))
and (⋃*A* ∈ *set L*. *output* (*actions A*)) − (⋃*A* ∈ *set L*. *input* (*actions A*)) =
output (*actions* (*asynComposition L*))
and (⋃*A* ∈ *set L*. *inner* (*actions A*)) ∪ ((⋃*A* ∈ *set L*. *input* (*actions A*)) ∩
(⋃*A* ∈ *set L*. *output* (*actions A*))) = *inner* (*actions* (*asynComposition L*))
apply *succeed*
apply (*insert* ‹*matchFSMList L*›)
apply (*frule asynCompositionValid*)
apply (*clarsimp simp*: *step-def asynComposition-def asynCompositionRaw-def*
Let-def asynCompositionValid)
apply (*subst input-access*, *rule matchFSMList-produces-actsig*, *assumption*)
apply (*rule refl*)
apply (*insert* ‹*matchFSMList L*›)
apply (*frule asynCompositionValid*)
apply (*clarsimp simp*: *step-def asynComposition-def asynCompositionRaw-def*
Let-def asynCompositionValid)

apply (*subst output-access*, *rule matchFSMList-produces-actsig*, *assumption*)
apply (*rule refl*)
apply (*insert* ‹*matchFSMList L*›)
apply (*frule asynCompositionValid*)
apply (*clarsimp simp*: *step-def asynComposition-def asynCompositionRaw-def Let-def asynCompositionValid*)
apply (*subst inner-access*, *rule matchFSMList-produces-actsig*, *assumption*)
by (*rule refl*)

lemma *composite-statespace.fold*:
assumes *matchFSMList L*
shows $(\textit{list-times-compr } L\ (\lambda A.\ \textit{states } A) \times \textit{powermultiset } ((\bigcup A{\in}\textit{set } L.\ \textit{inner } (\textit{actions } A)) \cup (\bigcup A{\in}\textit{set } L.\ \textit{input } (\textit{actions } A)) \cap (\bigcup A{\in}\textit{set } L.\ \textit{output } (\textit{actions } A)))) = \textit{states } (\textit{asynComposition } L)$
apply (*insert* ‹*matchFSMList L*›)
apply (*frule asynCompositionValid*)
by (*bestsimp simp*: *step-def asynComposition-def asynCompositionRaw-def Let-def asynCompositionValid*)

lemma *step-respects-statespace* [*rule-format*]:
shows $\textit{step } A\ q_1\ \textit{In Out } q_2 \longrightarrow q_1 \in \textit{states } A$
and $\textit{step } A\ q_1\ \textit{In Out } q_2 \longrightarrow q_2 \in \textit{states } A$
apply *succeed*
apply (*rule asynfsm.unfold*)
apply (*clarsimp simp*: *step-def steps-access actions-access*)
apply (*unfold asynfsm-def*, *clarsimp*)
apply (*erule-tac* $x = (q_1, \textit{In}, \textit{Out}, q_2)$ **in** *ballE-in*, *assumption*, *blast*)
apply (*rule asynfsm.unfold*)
apply (*clarsimp simp*: *step-def steps-access actions-access*)
apply (*unfold asynfsm-def*, *clarsimp*)
by (*erule-tac* $x = (q_1, \textit{In}, \textit{Out}, q_2)$ **in** *ballE-in*, *assumption*, *blast*)

lemma *list-times-compr.arbitrary-merging-update* [*intro*,*rule-format*]:
$[\![\textit{length } L_1 = \textit{length } L_2;\ \textit{length } L_2 = \textit{length } L_3;\ i < \textit{length } L_1;\ L_1 \in \textit{list-times-compr } L_3\ f;\ L_2 \in \textit{list-times-compr } L_3\ f]\!] \Longrightarrow L_1[i := L_2\ !\ i] \in \textit{list-times-compr } L_3\ f$

Consider a cross-product of length n of a list of sets. Now consider two tuples t and s out of this cross-product. Clearly switching some components between s and t will still lead to tuples within the cross-product.

by (*rule proofHole*[*of ?thesis*])

lemma *powermultiset-includes-subset*:
$[\![A \in \textit{powermultiset } S;\ B \subseteq\# A]\!] \Longrightarrow B \in \textit{powermultiset } S$
apply (*simp add*: *powermultiset-def*)
apply (*unfold set-of-def*)

apply ($subgoal\text{-}tac\ \forall x.\ x \in\#\ B \longrightarrow\ x \in S$, *blast*)
apply ($subgoal\text{-}tac\ \forall x.\ x \in\#\ A \longrightarrow\ x \in S$)
  prefer *2* **apply** *blast*
apply (*rule allI*)
apply (*erule-tac* $x = x$ **in** *allE*)
apply *clarsimp*
apply (*subgoal-tac* $x \in\#\ A$)
  apply *blast*
apply (*drule-tac* $x = x$ **and** $A = B$ **and** $B = A$ **in** *mset-leD*, *assumption*)
by *simp*

lemma *powermultiset-keeps-subset*: $[\![A \subseteq B]\!] \Longrightarrow$ *powermultiset* $A \subseteq$ *powermultiset* B
apply (*simp add*: *powermultiset-def*)
apply (*unfold set-of-def*)
by *blast*

lemma *powermultiset-contains-multiset-of*:
$[\![A \subseteq B]\!] \Longrightarrow$ *DrahflowTools.multiset-of* $A \in$ *powermultiset* B

Completely parallel to powersets.

by (*rule proofHole*[*of ?thesis*])

lemma *powermultiset-two-elements-implies-union*:
$[\![A \in$ *powermultiset* S; $B \in$ *powermultiset* $S]\!] \Longrightarrow A + B \in$ *powermultiset* S

Completely parallel to powersets.

by (*rule proofHole*[*of ?thesis*])

lemma *multiset-difference-subset-positive*: $A - S \subseteq\#\ A$
by (*rule proofHole*[*of ?thesis*])

lemma *confluence*:
assumes *compatibleMachines*: *matchFSMList* L
  and *serialMachines*: $\forall A \in$ *set* L. *serial* A
  and *parallelStep*: *step* (*asynComposition* L) (ql_1, M_1) *In Out* (ql_3, M_3)
  and *singleAction*: $i \in$ *In*
shows
  $\exists$ *Outi* ql_2 M_2. *step* (*asynComposition* L) (ql_1, M_1) $\{i\}$ *Outi* (ql_2, M_2) $\wedge$
  (*In* $= \{i\} \vee$ *step* (*asynComposition* L) (ql_2, M_2) (*In* $- \{i\}$) (*Out* $-$ *Outi*) (ql_3, M_3))
proof $-$
  from *compatibleMachines* **have** *validComposition*: *asynCompositionRaw* $L \in$ *asynfsm*
  by (*rule asynCompositionValid*)
  from *parallelStep* **and** *compatibleMachines* **and** *serialMachines*
  have *finiteActions*: *finite In* **by** *blast*

  from *parallelStep* **and** *validComposition*

obtain *Inl* **and** *Outl* **where**
let inputs = ((⋃ *A* ∈ *set L. input* (*actions A*)) − (⋃ *A* ∈ *set L. output* (*actions A*))) *in*
let outputs = ((⋃ *A* ∈ *set L. output* (*actions A*)) − (⋃ *A* ∈ *set L. input* (*actions A*))) *in*
let inners = ((⋃ *A* ∈ *set L. inner* (*actions A*)) ∪
(⋃ *A* ∈ *set L. input* (*actions A*) ∩ (⋃ *A* ∈ *set L. output* (*actions A*)))) *in*
let Q = ((*list-times-compr L* (λ*A. states A*)) × (*powermultiset inners*)) *in*
(
bool-and-map (λ(qi_1, *ini*, *outi*, qi_3, *Ai*).
((*step Ai* qi_1 *ini outi* qi_3 ∧ *multiset-of* (*ini* ∩ *input* (*actions Ai*) ∩ *inners*) ⊆# M_1) ∨
(*ini* = {} ∧ *outi* = {} ∧ $qi_1 = qi_3$)))
(*zip* ql_1 (*zip Inl* (*zip Outl* (*zip* ql_3 *L*)))) ∧
In = *set-aggr-filter* (*inputs* ∪ *inners*) *Inl* ∧ *In* ≠ {} ∧ *Out* = *set-aggr-filter outputs Outl* ∧
M_3 = (M_1 − *multiset-of In*) + *multiset-of* (*set-aggr-filter inners Outl*) ∧
(ql_1, M_1) ∈ *Q* ∧ (ql_3, M_3) ∈ *Q* ∧ *length* ql_1 = *length L* ∧ *length Inl* = *length L* ∧
length Outl = *length L* ∧ *length* ql_3 = *length L*
)
by (*bestsimp simp*: *step-def asynComposition-def asynCompositionRaw-def*
Let-def meta-allE[**where** *x* = *Inl*] *meta-allE*[**where** *x* = *Outl*])
note *conditionsOnInlAndOutl* = *this*

def *Outi-def*: *Outi* ≡ *Outl* ! *source-machine L i*
def *inners-def*: *inners* ≡ *inner* (*actions* (*asynComposition L*))

show *?thesis*
proof (*rule-tac x* = *Outi* **in** *exI*,
rule-tac x = ql_1[*source-machine L i* := ql_3 ! *source-machine L i*] **in** *exI*,
rule-tac x = M_1 − *multiset-of* {*i*} + *multiset-of* (*inners* ∩ *Outi*) **in** *exI*,
rule conjI)
from *validComposition*
show *step* (*asynComposition L*) (ql_1, M_1) {*i*} *Outi*
(ql_1[*source-machine L i* := ql_3 ! *source-machine L i*], M_1 −
DrahflowTools.multiset-of {*i*} + *DrahflowTools.multiset-of* (*inners* ∩ *Outi*))
proof (*clarsimp simp*: *step-def asynComposition-def asynCompositionRaw-def Let-def*,
rule-tac x = *replicate* (*length L*) {} [*source-machine L i* := {*i*}] **in** *exI*,
rule-tac x = *replicate* (*length L*) {} [*source-machine L i* := *Outi*] **in** *exI*)
let *?cond1* = *bool-and-map* (λ(qi_1, *ini*, *outi*, qi_2, *Ai*). (qi_1, *ini*, *outi*, qi_2) ∈ *steps Ai* ∧
DrahflowTools.multiset-of (*ini* ∩ *input* (*actions Ai*) ∩
((⋃*A*∈*set L. inner* (*actions A*)) ∪ (⋃*A*∈*set L. input* (*actions A*)) ∩
(⋃*A*∈*set L. output* (*actions A*)))) ⊆# M_1
∨ *ini* = {} ∧ *outi* = {} ∧ $qi_1 = qi_2$)
(*zip* ql_1 (*zip* (*replicate* (*length L*) {}[*source-machine L i* := {*i*}])
(*zip* (*replicate* (*length L*) {}[*source-machine L i* := *Outi*])
(*zip* (ql_1[*source-machine L i* := ql_3 ! *source-machine L i*]) *L*))))
let *?cond2* = {*i*} = *set-aggr-filter* ((⋃*A*∈*set L. input* (*actions A*)) −
(⋃*A*∈*set L. output* (*actions A*)) ∪ ((⋃*A*∈*set L. inner* (*actions A*)) ∪
(⋃*A*∈*set L. input* (*actions A*)) ∩ (⋃*A*∈*set L. output* (*actions A*))))
(*replicate* (*length L*) {}[*source-machine L i* := {*i*}])

let *?cond3* =
Outi = *set-aggr-filter* ((⋃*A*∈*set L*. *output* (*actions A*)) − (⋃*A*∈*set L*. *input* (*actions A*)))
(*replicate* (*length L*) {}[*source-machine L i* := *Outi*])
let *?cond4* = *DrahflowTools.multiset-of* (*inners* ∩ *Outi*) =
DrahflowTools.multiset-of (*set-aggr-filter*
((⋃*A*∈*set L*. *inner* (*actions A*)) ∪ (⋃*A*∈*set L*. *input* (*actions A*)) ∩
(⋃*A*∈*set L*. *output* (*actions A*)))
(*replicate* (*length L*) {}[*source-machine L i* := *Outi*]))
let *?cond5a* = ql_1 ∈ *list-times-compr L states*
let *?cond5b* = M_1 ∈ *powermultiset* ((⋃*A*∈*set L*. *inner* (*actions A*)) ∪
(⋃*A*∈*set L*. *input* (*actions A*)) ∩ (⋃*A*∈*set L*. *output* (*actions A*)))
let *?cond6a* =
ql_1[*source-machine L i* := ql_3 ! *source-machine L i*] ∈ *list-times-compr L states*
let *?cond6b* =
M_1 − *DrahflowTools.multiset-of* {*i*} + *DrahflowTools.multiset-of* (*inners* ∩ *Outi*)
∈ *powermultiset* ((⋃*A*∈*set L*. *inner* (*actions A*)) ∪
(⋃*A*∈*set L*. *input* (*actions A*)) ∩ (⋃*A*∈*set L*. *output* (*actions A*)))
let *?cond7* =
length ql_1 = *length L* ∧
length (*replicate* (*length L*) {}[*source-machine L i* := {*i*}]) = *length L* ∧
length (*replicate* (*length L*) {}[*source-machine L i* := *Outi*]) = *length L* ∧
length ql_1 = *length L*

from *compatibleMachines serialMachines parallelStep* **and** *singleAction*
have *?cond2*
apply (*subst set-aggr-filter.gobble-empty-replicate*, *rule source-machine-length*)
apply (*simp add*: *set-aggr-filter-def*)
apply (*subst composite-actions.fold*, *assumption*)+
apply (*frule step-respects-signature*(*2*))
by *blast*

moreover

from *compatibleMachines serialMachines parallelStep* **and** *singleAction*
have *?cond3*
apply (*subst set-aggr-filter.gobble-empty-replicate*, *rule source-machine-length*)
apply (*simp add*: *set-aggr-filter-def*)
apply (*subst composite-actions.fold*, *assumption*)+
apply (*frule step-respects-signature*(*2*))
apply (*unfold Outi-def*)

Clearly the source machine will only have emitted valid outputs.

by (*rule proofHole*[*of Outl* ! *source-machine L i* =
Outl ! *source-machine L i* ∩ *output* (*actions* (*asynComposition L*))])

moreover

```
from compatibleMachines serialMachines parallelStep and singleAction
have ?cond4
  apply (subst set-aggr-filter.gobble-empty-replicate, rule source-machine-length)
  apply (simp add: set-aggr-filter-def)
  apply (subst composite-actions.fold, assumption)+
  apply (unfold inners-def)
  apply (rule-tac f = multiset-of in eq-cong-fun-app)
by blast

moreover

from compatibleMachines and parallelStep
have ?cond5a ∧ ?cond5b
  apply (insert compatibleMachines)
  apply (insert parallelStep)
  apply (drule step-respects-statespace)
  apply (subgoal-tac (ql₁, M₁) ∈ list-times-compr L states ×
        powermultiset ((⋃A∈set L. inner (actions A)) ∪
          (⋃A∈set L. input (actions A)) ∩ (⋃A∈set L. output (actions A))))
    apply blast
by (subst composite-statespace.fold, assumption+)

moreover

from conditionsOnInlAndOutl have ?cond7 by (simp add: Let-def)

moreover

have ?cond6a ∧ ?cond6b
proof (rule conjI)
  show ql₁[source-machine L i := ql₃ ! source-machine L i] ∈ list-times-compr L states
    apply (insert conditionsOnInlAndOutl, simp add: Let-def)
    apply (rule list-times-compr.arbitrary-merging-update, simp, simp)
    proof −
      from compatibleMachines serialMachines parallelStep singleAction
      have source-machine L i < length L by (rule source-machine-length)
      moreover
      have length ql₁ = length L by (insert conditionsOnInlAndOutl, simp add: Let-def)
      ultimately
      show source-machine L i < length ql₁ by simp

      show ql₁ ∈ list-times-compr L states
      by (insert conditionsOnInlAndOutl, simp add: Let-def)
      show ql₃ ∈ list-times-compr L states
      by (insert conditionsOnInlAndOutl, simp add: Let-def)
    qed
```

```
    from conditionsOnInlAndOutl
    have M₁IsCorrect: M₁ ∈ powermultiset ((⋃A∈set L. inner (actions A)) ∪
            (⋃A∈set L. input (actions A)) ∩ (⋃A∈set L. output (actions A)))
    by (simp add: Let-def)

    moreover

    from compatibleMachines
    have newOutputIsCorrect: DrahflowTools.multiset-of (inners ∩ Outi)
        ∈ powermultiset ((⋃A∈set L. inner (actions A)) ∪
          (⋃A∈set L. input (actions A)) ∩ (⋃A∈set L. output (actions A)))
      apply (subst composite-actions.fold, assumption)+
      apply (unfold inners-def)
    by (blast intro: powermultiset-contains-multiset-of)

    ultimately
    show M₁ − DrahflowTools.multiset-of {i} +
      DrahflowTools.multiset-of (inners ∩ Outi) ∈ powermultiset
        ((⋃A∈set L. inner (actions A)) ∪
          (⋃A∈set L. input (actions A)) ∩ (⋃A∈set L. output (actions A)))
      apply (unfold inners-def)
      proof (rule powermultiset-two-elements-implies-union)
        show M₁ − DrahflowTools.multiset-of {i} ∈ powermultiset
                      ((⋃A∈set L. inner (actions A)) ∪
                        (⋃A∈set L. input (actions A)) ∩ (⋃A∈set L. output (actions A)))
          using M₁IsCorrect
          proof (rule powermultiset-includes-subset)
            show M₁ − DrahflowTools.multiset-of {i} ⊆# M₁
            by (rule multiset-difference-subset-positive)
          qed

        show DrahflowTools.multiset-of (inner (actions (asynComposition L)) ∩ Outi) ∈
                  powermultiset ((⋃A∈set L. inner (actions A)) ∪
                    (⋃A∈set L. input (actions A)) ∩ (⋃A∈set L. output (actions A)))
        by (insert newOutputIsCorrect, unfold inners-def, simp)
      qed
  qed

  moreover

  have ?cond1
    proof (rule bool-and-map.everyR, subst set-zip, rule ballI, fold step-def,
            clarsimp, rename-tac iPos)
      fix iPos
      show
        step (L ! iPos) (ql₁ ! iPos) (replicate (length L) {}[source-machine L i := {i}] ! iPos)
                  (replicate (length L) {}[source-machine L i := Outi] ! iPos)
```

```
            (ql1[source-machine L i := ql3 ! source-machine L i] ! iPos) ∧
        DrahflowTools.multiset-of
          (replicate (length L) {}[source-machine L i := {i}] ! iPos ∩
              input (actions (L ! iPos)) ∩
          ((⋃A∈set L. inner (actions A)) ∪
            (⋃A∈set L. input (actions A)) ∩ (⋃A∈set L. output (actions A)))) ⊆# M1
    proof (cases iPos = source-machine L i)
      case True
      show ?thesis
        apply (insert ‹iPos = source-machine L i›, insert conditionsOnInlAndOutl)
        apply (clarsimp simp: Let-def)
        apply (drule-tac list-times-compr-same-length)+
        apply (insert source-machine-length[of L (ql1, M1) In Out (ql3, M3) i])
        apply (simp add: compatibleMachines serialMachines parallelStep singleAction)
        apply (unfold Outi-def)
        apply (drule-tac bool-and-map.everyA)
        apply (erule-tac x = (ql1 ! source-machine L i, {i}, Outl ! source-machine L i,
            ql3 ! source-machine L i, L ! source-machine L i) in ballE-in)
          apply (insert source-machine-length[of L (ql1, M1) In Out (ql3, M3) i])
          apply (simp add: compatibleMachines serialMachines parallelStep singleAction)
          apply (unfold set-zip)
          apply clarsimp
          apply (rule-tac x = source-machine L i in exI, clarsimp)
          apply (insert singleAction conditionsOnInlAndOutl)
          apply (clarsimp simp: Let-def)
          apply (drule bool-and-map.everyA)
          apply (rule directContradiction)
          apply (insert source-machine-input[of L (ql1, M1) In Out (ql3, M3) i])
          apply (simp add: compatibleMachines serialMachines parallelStep singleAction)
```

From $\{i\} \neq \mathit{Inl}$! *source-machine* L i and $i \in$ *input* (*actions* (L ! *source-machine* L i)) $\vee$ $i \in$ *inner* (*actions* (L ! *source-machine* L i)) follows a contradiction.

```
          apply (rule proofHole[of False])

        apply (insert source-machine-length[of L (ql1, M1) In Out (ql3, M3) i])
      by (simp add: compatibleMachines serialMachines parallelStep singleAction)

      next

      case False
      show ?thesis
```

This case can never occur, as all input and inner signatures are disjunct.

```
      by (rule proofHole[of ?thesis])
    qed
  qed
```

```
    ultimately
    show ?cond1 ∧ ?cond2 ∧ ?cond3 ∧ ?cond4 ∧ ?cond5a ∧ ?cond5b ∧
      ?cond6a ∧ ?cond6b ∧ ?cond7
    by blast
  qed

  next
  show In = {i} ∨ step (asynComposition L)
    (ql1[source-machine L i := ql3 ! source-machine L i],
      M1 − DrahflowTools.multiset-of {i} + DrahflowTools.multiset-of (inners ∩ Outi))
    (In − {i}) (Out − Outi) (ql3, M3)
  proof (cases In = {i})
    case True thus ?thesis by blast
    next
    case False
    with validComposition
    show ?thesis
    proof (clarsimp simp: step-def asynComposition-def asynCompositionRaw-def Let-def,
          rule-tac x = Inl[source-machine L i := {}] in exI,
          rule-tac x = Outl[source-machine L i := {}] in exI)
      let ?cond1 = bool-and-map (λ(qi1, ini, outi, qi2, Ai). (qi1, ini, outi, qi2) ∈ steps Ai ∧
                     DrahflowTools.multiset-of (ini ∩ input (actions Ai) ∩
                       ((⋃A∈set L. inner (actions A)) ∪
                        (⋃A∈set L. input (actions A)) ∩ (⋃A∈set L. output (actions A))))
                        ⊆# M1 − DrahflowTools.multiset-of {i} +
                          DrahflowTools.multiset-of (inners ∩ Outi) ∨
                     ini = {} ∧ outi = {} ∧ qi1 = qi2)
                   (zip (ql1[source-machine L i := ql3 ! source-machine L i])
                    (zip (Inl[source-machine L i := {}]) (zip (Outl[source-machine L i := {}])
                     (zip ql3 L))))
      let ?cond2 = In − {i} = set-aggr-filter ((⋃A∈set L. input (actions A)) −
                     (⋃A∈set L. output (actions A)) ∪
                       ((⋃A∈set L. inner (actions A)) ∪
                       (⋃A∈set L. input (actions A)) ∩ (⋃A∈set L. output (actions A))))
                     (Inl[source-machine L i := {}])
      let ?cond3 = ¬ In ⊆ {i}
      let ?cond4 = Out − Outi = set-aggr-filter ((⋃A∈set L. output (actions A)) −
                     (⋃A∈set L. input (actions A)))
                     (Outl[source-machine L i := {}])
      let ?cond5 = M3 = M1 − DrahflowTools.multiset-of {i} +
                 DrahflowTools.multiset-of (inners ∩ Outi) −
                 DrahflowTools.multiset-of (In − {i}) + DrahflowTools.multiset-of
                  (set-aggr-filter
                    ((⋃A∈set L. inner (actions A)) ∪
                     (⋃A∈set L. input (actions A)) ∩ (⋃A∈set L. output (actions A)))
                    (Outl[source-machine L i := {}]))
```

let *?cond6a* = ql_1[*source-machine L i* := ql_3 ! *source-machine L i*] ∈
list-times-compr L states
let *?cond6b* = M_1 − *DrahflowTools.multiset-of* {*i*} +
DrahflowTools.multiset-of (*inners* ∩ *Outi*) ∈ *powermultiset*
((⋃*A*∈*set L*. *inner* (*actions A*)) ∪
(⋃*A*∈*set L*. *input* (*actions A*)) ∩ (⋃*A*∈*set L*. *output* (*actions A*)))
let *?cond7a* = ql_3 ∈ *list-times-compr L states*
let *?cond7b* = M_3 ∈ *powermultiset* ((⋃*A*∈*set L*. *inner* (*actions A*)) ∪
(⋃*A*∈*set L*. *input* (*actions A*)) ∩ (⋃*A*∈*set L*. *output* (*actions A*)))
let *?cond8* = *length* ql_1 = *length L* ∧ *length* (*Inl*[*source-machine L i* := {}]) = *length L* ∧
length (*Outl*[*source-machine L i* := {}]) = *length L* ∧
length ql_3 = *length L*

In principle parallel to the above proof about the single element.

have *?cond1* **by** (*rule proofHole*[*of ?thesis*])
moreover
have *?cond2* **by** (*rule proofHole*[*of ?thesis*])
moreover
have *?cond3* **by** (*rule proofHole*[*of ?thesis*])
moreover
have *?cond4* **by** (*rule proofHole*[*of ?thesis*])
moreover
have *?cond5* **by** (*rule proofHole*[*of ?thesis*])
moreover
have *?cond6a* ∧ *?cond6b* **by** (*rule proofHole*[*of ?thesis*])
moreover
have *?cond7a* ∧ *?cond7b* **by** (*rule proofHole*[*of ?thesis*])
moreover
have *?cond8* **by** (*rule proofHole*[*of ?thesis*])
ultimately
show *?cond1* ∧ *?cond2* ∧ *?cond3* ∧ *?cond4* ∧ *?cond5* ∧ *?cond6a* ∧ *?cond6b* ∧
?cond7a ∧ *?cond7b* ∧ *?cond8*
by *blast*
qed
qed
qed
qed

lemma *confluence-corollary*:
⟦*matchFSMList L*; ⋀*x*. *x* ∈ *set L* ⟹ *serial x*; *P* q_1;
⋀q_1 *i Out* q_2. ⟦*P* q_1; ⋀*Out* q_2. *step* (*asynComposition L*) q_1 {*i*} *Out* q_2⟧ ⟹ *P* q_2⟧
⟹ *step* (*asynComposition L*) q_1 *In Out* q_2 ⟶ *P* q_2

Via induction over the set In, taking one action out at a time always carrying along P.

by (*rule proofHole*[*of ?thesis*])

inductive-set *reachable* :: $('q,'act)asynfsm \Rightarrow 'q\ set$ **for** A :: $('q,'act)asynfsm$
where *initial* $A \in$ *reachable* A
and $[\![q \in \mathit{reachable}\ A; \exists\, \mathit{In}\ \mathit{Out}.\ \mathit{step}\ A\ q\ \mathit{In}\ \mathit{Out}\ q']\!] \Longrightarrow\ q' \in \mathit{reachable}\ A$

lemma *confluence-invariant*:
$[\![\mathit{matchFSMList}\ L; \bigwedge x.\ x \in \mathit{set}\ L \Longrightarrow \mathit{serial}\ x; q \in \mathit{reachable}\ (\mathit{asynComposition}\ L);$
$P\ (\mathit{initial}\ (\mathit{asynComposition}\ L));$
$\bigwedge q_1\ i\ \mathit{Out}\ q_2.\ [\![P\ q_1; \mathit{step}\ (\mathit{asynComposition}\ L)\ q_1\ \{i\}\ \mathit{Out}\ q_2]\!] \Longrightarrow P\ q_2]\!]$
$\Longrightarrow P\ q$
apply (*erule-tac reachable.induct*, *assumption*)
apply (*erule-tac exE*)+
by (*drule-tac* $q_1 = q$ **and** $q_2 = q'$ **and** $L = L$ **and** $P = P$ **and** $In = In$ **and** $Out = Out$
in *confluence-corollary*, *blast*+)

end

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Braunschweig, 6. Januar 2009