A Refinement Based Strategy for Local Deadlock Analysis of Networks of CSP Processes — Extended version

Pedro Antonino, Augusto Sampaio
Universidade Federal de Pernambuco,
Centro de Informática, Recife, Brazil
{prga2,acas}@cin.ufpe.br

Jim Woodcock
University of York,
Department of Computer Science, York, UK
jim.woodcock@york.ac.uk

December 11, 2013

Abstract

Based on a characterisation of process networks in the CSP process algebra, we formalise a set of behavioural restrictions used for local deadlock analysis. Also, we formalise two patterns, originally proposed by Roscoe, which avoid deadlocks in cyclic networks by performing only local analyses on components of the network; our formalisation systematises the behavioural and structural constraints imposed by the patterns. A distinguishing feature of our approach is the use of refinement expressions for capturing notions of pattern conformance, which can be mechanically checked by CSP tools like FDR. Moreover, three examples are introduced to demonstrate the effectiveness of our strategy, including a performance comparison between FDR default deadlock assertion and the verification of local behavioural constraints induced by our approach, also using FDR.

1 Introduction

There are a number of ways to prove that a system is deadlock free. One approach is to prove, using a proof system and semantic model, that a deadlock state is not reachable [10]. Another approach is to model check a system in order to verify that a deadlock state cannot be reached [9]. Both approaches have substantial drawbacks. Concerning the first approach, it is not fully automatic and requires one to have a vast knowledge of: the semantic model, the notation employed in the model and the proof system used. In the second approach, although automatic, deadlock verification can became unmanageable due to the exponential growth with the number of components of the system. To illustrate these problems, let us assume that one is trying to prove that the dinning philosophers is deadlock free using the CSP notation [5, 9, 12]. In the first approach, one must be familiar with the *stable failures* semantic model [3, 9, 12] and with a proof system to carry the proof itself. In the second case, assuming that we have philosopher and fork processes with 7 and 4 states, respectively, the number of states can grow up to $7^N \times 4^N$, where N is the number of philosophers in the configuration. For instance, to verify that a system with 50 philosophers and 50 forks is deadlock free one has to verify up to $7^{50} \times 4^{50}$ states.

One alternative to these approaches is to adopt a hybrid technique, which consists of proving, using semantic models and a proof system, that for a particular class of well-defined systems, a property can be verified by only checking a small portion of the system. This principle, called *local analysis*, is the core technique of some existing approaches to compositional analysis [1, 2]. Concerning deadlock analysis, in particular, the strategy reported in [10, 4] introduces a network model and behavioural constraints that support local analysis.

Nevertheless, despite the provided conceptual support for local deadlock analysis, no automated strategy is available. Our approach provides a detailed formalisation of the network model and behavioural constraints presented in [10, 4], from which refinement assertions that can be checked using FDR are derived. Also, we formalise two patterns for deadlock avoidance, together with refinement assertions that automatically ensures adherence to the patterns.

Finally, three examples are introduced as a proof of concept of our refinement based strategy, as well as a performance comparison between our strategy and the FDR [13] deadlock freedom assertion.

In the next section we briefly introduce CSP. In Section 3 we present the network model [10, 4] on which we base our approach. Our major contributions are presented in Section 4: the formalisation of a behavioural condition that guarantees deadlock freedom for acyclic network, the formalisation of two communication patterns that avoid deadlocks in acyclic networks, and a refinement based technique for verifying behavioural constraints of the network model and conformance to the patterns. Section 5 provides practical evaluation and Section 6 gives our conclusions, as well as related and future work.

2 CSP

CSP is a process algebra that can be used to describe systems as interacting components, which are independent self-contained processes with interfaces that are used to interact with the environment [9]. Most of the CSP tools, like FDR, accept a machine-processable CSP, called CSP_M, used in this paper.

The two basic CSP processes are `STOP` and `SKIP`; the former deadlocks, and the latter does nothing and terminates. The prefixing `a -> P` is initially able to perform only the event `a`; afterwards it behaves like process `P`. The alternation `if b then P else Q` is available and has a standard behaviour. The operator `P1;P2` combines `P1` and `P2` in sequence. The external choice `P1[]P2` initially offers events of both processes; the occurrence of the first event or termination resolves the choice in favour of the process that performs either of them. The environment has no control over the internal choice `P1|~|P2`, in which the choice is resolved internally. The alphabetised parallel composition `P1[cs1||cs2]P2` allows `P1` and `P2` to communicate in the sets `cs1` and `cs2`, respectively; however, they must agree on events in `cs1`$\cap$`cs2`. The event hiding operator `P\cs` encapsulates the events that are in `cs`. The renamed process `P[[a<-b]]` behaves like `P` except that all occurrences of `a` in `P` are replaced by `b`; the relational renaming, when there is a relation of many new event to an old one, as in `P[a <- b, a <- c]`, results in the extension of the behaviour by allowing the process to offer deterministically both `b` and `c` when it offered a.

CSP also provides replicated versions for most of its compositional operators. For instance, `PP = || x :  S @ [A(x)] P(x)` stands for the alphabetised parallel composition of all `P(x)` using `A(x)` as its alphabet, for `x`$\in$`S`. Local processes are defined using the `let Id1 = P1, ..., Idk = Pk within Q` construct, which behaves as `Q` and restricts the scope of the processes `Id1, ..., Idk` to `Q`.

Two CSP semantic models are used in this work: the *stable failures*, and the *stable-revivals* models [12]. In the *stable failures* model, a process is represented by its traces, which is a set of finite sequences of events it can perform, given by *traces*(P), and by its stable failures. Stable failures are pairs (s, X) where s is a finite trace and X is a set of events that the process can refuse to do after performing the trace s. At the state where the process can refuse events in X, the process must not be able to perform an internal action, otherwise this state would be unstable and would not be taken into account in this model. The function $refusals(P, s)$ gives the set of X's that a process P can refuse after s, and *failures*(P) gives the set of stable failures of process P. The stable revivals model has three components: traces, deadlocks and revivals. The *traces* component is the same one as that described for the other models. The *deadlocks* component gives the set of traces after which the process deadlocks. Finally, the *revivals* component gives the set of triples (s, X, a) which is composed of a trace s of the process, a set of refusals X after this trace, and an event that can be performed after this refusal a, the revival event.

For each model, there is a refinement relation given by `[M=`. `M` can be `T`,`F` or `V` for traces, stable failures and stable revivals, refinement relation respectively. The refinement expression `P [M= Q` holds if and only if for each component of model `M`, *component*$(P) \supseteq$ *component*(Q). For instance, for the stable failures model, `P [V= Q` $\Leftrightarrow failures(P) \supseteq failures(Q) \wedge traces(P) \supseteq traces(Q)$.

The choice of a model involves considerations about the semantic domain convenient to capture the relevant property. The properties that can only be expressed in terms of maximal failures are more intuitively represented in the stable revivals model, since this model carries partial information about the maximal failure: the revival event. On the other hand, the restrictions that can be expressed without being confined to maximal failures can be easily captured by the stable failure model and its refinement relation.

3 Network model

The concepts presented in this section are essentially drawn from [4, 10], which present an approach to deadlock analysis of systems described as a network of CSP processes. The most fundamental concept is the one of *atomic tuples*, which represents the basic components of a system. These are triples that contain an identifier for the component, the process describing the behaviour of this component and an alphabet that represents the set of events that this component can perform. A *network* is a finite set of atomic tuples.

Definition 1 (Network)**.** *Let $CSP\_Processes$ be the set of all possible CSP processes, Σ the set of CSP events and $IdType$ the set for identifiers of atomic tuples. A network is a set V, such that:*

$$V \subset Atomics$$

where: $Atomics \mathrel{\widehat=} IdType \times \mathcal{P}\Sigma \times CSP\_Processes$ and V is finite

The behaviour of a network is given as a composition of the behaviour of each component using the CSP alphabetised parallel operator, where the behaviour and alphabet from the atomic tuple identified by id are extracted by the functions $B(id, V)$ and $A(id, V)$ respectively. We use the indexed version of the alphabetised parallel operator.

Definition 2 (Behaviour of a network)**.** *Let V be a network.*
$B(V) \mathrel{\widehat=}$ `|| id : dom V @ [A(id,V)] B(id,V)`

A *live* network is a structure that satisfies three assumptions. The first one is *busyness*. A busy network is a network whose atomic components are deadlock free. The second assumption is *atomic non-termination*, i.e. no atomic component can terminate. The last assumption concerns interactions. A network is *triple-disjoint* if at most two processes share an event, i.e. if for any three different atomic tuples their alphabet intersection is the empty set.

In a *live* network, a deadlock state can only arise from an improper interaction between processes, since no process can individually deadlock. This particular misinteraction is captured by the concept of *ungranted requests*. An ungranted request occur in a particular state $\sigma = (s, R)$ of the network. In this state, s is a trace of the network and R is a vector of refusal sets, $R(id)$ being the refusal set of the process id after $s \upharpoonright A(id, V)$, where $s \upharpoonright A(id, V)$ corresponds to trace s restricted to events in $A(id, V)$. We introduce the notations $\sigma.s$ and $\sigma.R$ to get the s and the R component of state σ, respectively. An ungranted request arises in a state σ when an atom, say id_1, is offering an event to communicate with another atom, say id_2, but id_2 cannot offer any of the events expected by id_1. In addition, both processes must not be able to perform internal actions, i.e. events that do not involve the synchronisation with another process.

Definition 3 (Ungranted request)**.** *Let id_1 and id_2 be identifiers of processes in a network V, $A_1 = A(id_1, V)$, $A_2 = A(id_2, V)$ and $Voc(V)$ the set of shared* *events of network V. There is an ungranted request from id_1 to id_2 in state σ if the following predicate holds:*

$$
\begin{aligned}
& ungranted\_request(V, \sigma, id_1, id_2) \mathrel{\widehat{=}} \\
& \quad request(V, \sigma, id_1, id_2) \wedge ungrantedness(V, \sigma, id_1, id_2) \\
& \quad \wedge in\_vocabulary(V, \sigma, id_1, id_2)
\end{aligned}
$$

- $request(V, \sigma, id_1, id_2) \mathrel{\widehat{=}} (A_1 \setminus \sigma.R(id_1)) \cap A_2 \neq \emptyset$
- $ungrantedness(V, \sigma, id_1, id_2) \mathrel{\widehat{=}} (A_1 \cap A_2) \subseteq (\sigma.R(id_1) \cup \sigma.R(id_2))$
- $in\_vocabulary(V, \sigma, id_1, id_2) \mathrel{\widehat{=}} (A_1 \setminus \sigma.R(id_1)) \cup (A_2 \setminus \sigma.R(id_2)) \subseteq Voc(V)$

Ungranted requests are the building blocks of a more complex structure denoted cycle of ungranted requests. A cycle of this kind is represented as a sequence of different process identifiers, C, where each element at the position i, $C(i)$, has an ungranted request to the element at the position $i \oplus 1$, $C(i \oplus 1)$, where $\oplus$ is addition modulo length of the sequence. A *conflict* is a proper cycle of ungranted requests with length 2. After these definitions a fundamental theorem extracted from [4] is introduced.

Theorem 1. *Let V be a live network. Any deadlocked state has a cycle of ungranted requests.*

Theorem 1 allows one to reduce the problem of avoiding deadlock by preventing cycles of ungranted requests. With this result it is already possible to fully verify a tree topology network in a local way, by checking only pairs of processes, due to the fact that only conflicts can arise in tree networks. Nevertheless, networks with cycles in their topology cannot be locally verified by this method, since the verification of absence of cycles of ungranted requests with length greater than 2 involves a global verification of the entire system.

Also we present a rule (theorem) drawn from [10], stating allows one to reduce the task of guaranteeing deadlock freedom to the task of finding a set of functions on the semantics of processes, $g(\sigma, i)$ and a ordering relation such that when there is a request from atom id_1 to atom id_2, $g(\sigma, id_2) > g(\sigma, id_2)$. This is formalised as follows.

Theorem 2 (Ordering ungranted requests)**.** *Let V be a network and that $(\Pi, >)$ is a strict partial order. Then if the functions $g(\sigma, id)$ have the property that, whenever σ is a state of any two-element subnetwork having the identifiers id_1 and id_2 where $id_1 \neq id_2$*

$$ungranted\_request(\sigma, id_1, id_2, V) \Rightarrow g(\sigma, id_1) > g(\sigma, id_2).$$

Then V is deadlock free.

In [4, 9, 10, 6], a set of patterns and examples of classes of networks is defined by semantic behavioural properties and a rather informal description of the their network structure. Although helpful for designing deadlock free systems, these patterns lack systematisation, and more importantly, the associated restrictions are expressed as semantic properties that must be proved in a semantic model. Also, some of the properties are too restrictive; for instance, the behaviour of a resource process is tied to be the one given by the rule. As a major contribution of this work, we present an approach to fully systematise and formalise these patterns. Also, we derive refinement assertions that precisely capture the conformance to a particular pattern. Two examples are provided.

4 Local deadlock analysis based on Patterns and Refinement Checking

In the approach for avoiding deadlock presented here we derive refinement expressions to capture behavioural properties. Besides the induced systematisation, these expressions can be verified using a refinement checker, enabling one to automatically verify behavioural constraints.

The first concept that we present is a function used to abstract the behaviour that is insignificant for deadlock analysis. If a process of a network can perform an individual event in a state σ, i.e., an event that does not require the permission of another process, then this state is deadlock free, since this process can perform this event. Thus, for the purpose of deadlock analysis, all states where a process offer an individual event can be discarded as deadlock is impossible. As we are not concerned with divergent behaviour, the hiding operator is used to abstract this meaningless states.

Definition 4 (Abstraction function)**.** *For a network V, let $B(id, V)$ be the behaviour, $A(id, V)$ the alphabet and $AVoc(id, V)$ the set of events used for communicating with other processes of atom id. Then we define:*

```
Abs(id,V) = B(id,V) \ diff(A(id,V),AVoc(id,V))
```

where: `AVoc(id,V) = Union({inter(A(id,V),A(ID_(a),V)) | a <- V, ID_(a) != id})`

A conflict is another concept of interest in deadlock analysis. As already discussed, it allows one to locally verify an acyclic network to be deadlock free. Conflict can be more intuitively captured by a refinement expression if the pair of atoms being verified for conflict is placed in a particular behavioural context. This context first abstracts the behavior of both atoms by using the function `Abs` and extend their behaviour by allowing them to deterministically offer the special event *req* whenever an event from $A(id1, V) \cap A(id2, V)$ is offered. Secondly, it composes the pair of processes using the alphabets extended with the *req* event. This context is given by the `Context` process, where the `Ext` process performs the abstraction and extension mentioned.

Definition 5 (Extended behaviour of a pair of processes)**.** *Let $id1$ and $id2$ be two processes of network V.*

```
Context(id1,id2,V)= Ext(id1,id2,V)[union(A(id1,V),{req})||union(A(id2,V),{req})]Ext(id2,id1,V)
```

where: `Ext(id1,id2,V) = Abs(id1,V) [[ x <- x, x <- req | x <- inter(A(id1,V),A(id2,V))]]`

When placed in this context, a conflict arises when the *req* event is offered and $A(id1, V) \cap A(id2, V)$ is refused. Hence, a conflict free pair of processes does not have a revival of the form (s, X, req) where $A(id_1) \cap A(id_1) \subseteq X$. The process `ConflictFreeSpec`, presented next, describes a process that has every possible behaviour but the ones that generate the conflicting form of revivals. It specifies all the states such that when *req* is offered, then $A(id1, V) \cap A(id2, V)$ is not refused. The `Context` is conflict free, if the following refinement expression holds.

Definition 6 (Extended behavior conflict freedom specification)**.** *Let* $id1$ *and* $id2$ *be two identifiers of atoms of network* V*.*

```
ConflictFreeSpec(id1,id2,V) =
let U_A = union(A(id1,V),A(id2,V))
    I_A = inter(A(id1,V),A(id2,V))
    CF_ = ((|~| ev : I_A @ ev -> CF_) [] req -> CHAOS(union(U_A,{req})))
          |~| (|~| ev : U_A @ ev -> CF_)
within CF_
```

where: `CHAOS(Alp) = SKIP |~| STOP |~| (|~| ev : Alp @ ev -> CHAOS(Alp))`

Definition 7 (Conflict freedom predicate)**.** *Let* id_1 *and* id_2 *be two identifiers of network* V*.*

$$
\begin{aligned}
&ConflictFree(id_1, id_2, V) \mathrel{\widehat=} \\
&\qquad \forall\, \sigma \bullet state(\sigma, V) \Rightarrow \neg conflict(\sigma, id_1, id_2, V)
\end{aligned}
$$

Theorem 3 (ConflictFreedomSpec stable revivals)**.**

$$
\begin{aligned}
&traces(ConflictFreeSpec(id_1, id_2, V)) \mathrel{\widehat=} (A(id_1, V) \cup A(id_2, V) \cup \{req\})^* \\
&deadlocks(ConflictFreeSpec(id_1, id_2, V)) \mathrel{\widehat=} \{s | req \in \operatorname{dom} s\} \\
&revivals(ConflictFreeSpec(id_1, id_2, V)) \mathrel{\widehat=} \\
&\qquad \{(s, X, a) | s \in (A_1 \cup A_2 \cup \{req\})^* \wedge \\
&\qquad a \in (A_1 \cup A_2 \cup \{req\}) \wedge a \notin X \wedge \\
&\qquad (a = req \Rightarrow (A_1 \cap A_2) \not\subseteq X)\}
\end{aligned}
$$

Proof. Calculated with the revivals, deadlocks and traces clauses. □

Theorem 4 (Context stable revivals)**.**

$$
\begin{aligned}
&traces(ConflictFreeSpec(id_1, id_2, V)) \mathrel{\widehat=} (A(id_1, V) \cup A(id_2, V) \cup \{req\})^* \\
&deadlocks(ConflictFreeSpec(id_1, id_2, V)) \mathrel{\widehat=} \{s | req \in \operatorname{ran} s\} \\
&revivals(ConflictFreeSpec(id_1, id_2, V)) \mathrel{\widehat=} \\
&\qquad \{(s, X, a) | s \in (A_1 \cup A_2 \cup \{req\})^* \wedge \\
&\qquad a \in (A_1 \cup A_2 \cup \{req\}) \wedge a \notin X \wedge \\
&\qquad (a = req \Rightarrow (A_1 \cap A_2) \not\subseteq X)\}
\end{aligned}
$$

Proof. Calculated with the revivals, deadlocks and traces clauses. □

We prove the soundness of our specification by the following theorem.

Theorem 5 (Soundness of conflict freedom refinement expression)**.** *Let* $V = \{(id1, B1, A1), (id2, B2, A2)\}$
`ConflictFreeSpec(id1,id2,V) [V=Context(id1,id2,V)` $\iff ConflictFree(id1, id2, V)$.

Proof. Let $ConflictFreeSpec(id1, id2, V) = CFS$, $Context(id1, id2, V) = Cx$ and $ConflictFree(id_1, id_2, V) \equiv CF$.
First case($\Longrightarrow$):

CFS `[V=` $Cx \wedge \neg CF$ [Assumption]
$\Longrightarrow$ [`[V=` and CF defs]
$revivals(Cx) \subseteq revivals(CFS) \wedge$
$(\exists \sigma \bullet state(\sigma, V) \wedge conflict(\sigma, id1, id2, V))$
$\Longrightarrow$ [$conflict$ def]
$revivals(Cx) \subseteq revivals(CFS) \wedge$
$(\exists \sigma \bullet state(\sigma, V) \wedge$
$request(\sigma, id1, id2, V) \wedge$
$request(\sigma, id2, id1, V) \wedge$
$ungrantedness(\sigma, id1, id2, V) \wedge$
$in\_vocabulary(\sigma, id1, id2, V))$
$\Longrightarrow$ [$revivals(Cx)$ def]
$revivals(Cx) \subseteq revivals(CFS) \wedge$
$(\sigma.s, A(id1, V) \cup A(id2, V), req) \in revivals(Cx)$
$\Longrightarrow$ [PC and ST]
$false$
$\Longrightarrow$ [PC]
CFS `[V=` $Cx \Rightarrow CF$

Other direction($\Longleftarrow$):

CF [As 1]
$\forall \sigma \bullet state(\sigma, V) \Rightarrow \neg conflict(\sigma, V)$ [CF def]
$\forall \sigma \bullet state(\sigma, V) \Rightarrow$ [$conflict$ def]
$(\neg request(\sigma, id1, id2, V) \vee$
$\neg request(\sigma, id2, id1, V) \vee$
$\neg ungrantedness(\sigma, id1, id2, V) \vee$
$\neg in\_vocabulary(\sigma, id1, id2, V))$

Case 1:

$\forall \sigma \bullet state(\sigma, V) \Rightarrow$
$(\neg request(\sigma, id1, id2, V) \vee$
$\neg request(\sigma, id2, id1, V))$

$\Longrightarrow$ [Cx stable revival semantics and $request$ def]

$traces(Cx) \subseteq (A_1 \cup A_2)^* \wedge$
$revivals(Cx) \subseteq \{(s, X, a) | s \in (A_1 \cup A_2)^* \wedge a \in (A_1 \cup A_2) \wedge a \notin X\} \wedge$
$deadlocks(Cx) = \emptyset$

$\Longrightarrow$ [ST]

$traces(Cx) \subseteq traces(CFS) \wedge$
$revivals(Cx) \subseteq revivals(CFS) \wedge$
$deadlocks(Cx) \subseteq deadlocks(CFS)$

$\Longrightarrow$ [`[V=` def]

CFS `[V=` Cx

Case 2:

$\forall \sigma \bullet state(\sigma, V) \Rightarrow$
$\neg ungrantedness(\sigma, id1, id2, V)$

$\Longrightarrow$ [$ungrantedness$ and stable revivals semantics of Cx def]

$traces(Cx) \subseteq (A_1 \cup A_2 \cup req)^* \wedge$
$revivals(Cx) \subseteq \{(s, X, a) | s \in (A_1 \cup A_2 \cup req)^* \wedge a \in (A_1 \cup A_2 \cup req) \wedge$
$\quad a \notin X \wedge (A_1 \cup A_2) \not\subseteq X\} \wedge$
$deadlocks(Cx) = \{s | req \in \operatorname{ran} s\}$

$\Longrightarrow$ [ST]

$traces(Cx) \subseteq traces(CFS) \wedge$
$revivals(Cx) \subseteq revivals(CFS) \wedge$
$deadlocks(Cx) \subseteq deadlocks(CFS)$

$\Longrightarrow$ [`[V=` def]

CFS `[V=` Cx

Case 3:

$\forall \sigma \bullet state(\sigma, V) \Rightarrow$
$\neg in\_vocabulary(\sigma, id1, id2, V)$

$\Longrightarrow$ [Cx stable revival semantics and $in\_vocabulary$ def]

$traces(Cx) \subseteq (A_1 \cup A_2 \cup req)^* \wedge$
$revivals(Cx) = \emptyset$
$deadlocks(Cx) = \emptyset$

$\Longrightarrow$ [ST]

$traces(Cx) \subseteq traces(CFS) \wedge$
$revivals(Cx) \subseteq revivals(CFS) \wedge$
$deadlocks(Cx) \subseteq deadlocks(CFS)$

$\Longrightarrow$ [`[V=` def]

CFS `[V=` Cx

$\square$

4.1 Behavioral patterns

In this section we introduce a set of patterns that prevent deadlocks for cyclic networks. They are based on design rules and class of networks presented in the literature. Nevetheless, in this work we systematise the conditions that must hold in order to a network be compliant to a pattern. Moreover, we introduce a way of capturing behavioural restrictions though refinement expressions that is new to the best knowledge of the author.

This refinement based technique relies on a particular aspect of the maximal failures of a refined process. Let us call the process on the left hand side of the refinement relation the abstract process, and the one on the right hand the concrete one. If the refinement relation holds, the maximal failures of the concrete process must lie within the set of failures such that, let f be a failure of this set, then f must refuse the set of impossible events after $f.s$. If the maximal failure lies outside this set, then the refinement relation does not hold, since either the failures restriction is violated or the traces one. This result is stated in the next theorem.

Theorem 6 (Maximal failures induced by refinement)**.** *Let P and Q be two arbitrary processes.*

$$P \sqsubseteq_F Q \Rightarrow Mfailures(Q) \subseteq MCfailures(P)$$

where: $MCfailures(P) \mathrel{\widehat{=}} \{f : failures(P) | f.R \supseteq \overline{initials(P/f.s)}^{\Sigma}\}$

Proof. This proof can be found in Appendix A. □

In a very similar manner, as we use the stable revival model for a behavioural restriction on the Client/Server pattern, we also demonstrate that if the refinement holds then there is a specific set of failures of P within which the maximal revivals of Q must lie.

Theorem 7 (Maximal revivals induced by stable revival refinement)**.** *Let P and Q be two deadlock free processes.*

$P \sqsubseteq_V Q \Rightarrow Mrevivals(Q) \subseteq MCrevivals(P)$

where: $Mrevivals(Q) \mathrel{\widehat{=}} \{r | r \in revivals(Q) \wedge max(r, Q)\}$
$MCrevivals(Q) \mathrel{\widehat{=}} \{r | r \in revivals(Q) \wedge r.R \supseteq \overline{initials(failure(r))}^{\Sigma} \wedge r.R \supseteq \overline{initials(r.s)}^{\Sigma}\}$

Proof. This proof can be found in Appendix A. □

4.1.1 Resource allocation pattern

The resource allocation pattern can be applied to systems that, in order to perform an action, have to acquire some shared resources such as a lock. In this pattern the atoms of a network are divided into *user* and *resource* processes. The functions $acquire(id_U, id_R)$ and $release(id_U, id_R)$ give the event used by the user process id_U to acquire (and, recpectively, release) the resource id_R. This pattern imposes a behavioural restriction on both resource and user processes.

The expected behaviour of a resource is given by the following process. It offers the events of acquisition to all users able to acquire this resource and, once acquired, it offers the release event to the user that has acquired it.

Definition 8 (Resource specification)**.** *Let id be an identifier of a resource atom and $users(id)$ a set of user identifiers used by this resource.*

```
ResourceSpec(id,V) =
    let idsU = users(id)
        Resource =
            [] idU : idsU @
                acquire(idU,id) -> release(idU,id) -> Resource
    within Resource
```

The required behaviour of a user is given by the following process. It first acquires all the necessary resources and then releases them. Both acquiring and releasing must be performed using the order denoted by the $resources(id)$ sequence.

Definition 9 (User specification)**.** *Let id be an identifier of a user atom and $resources(id)$ a sequence of resource identifiers in which this user atom acquire its resources.*

```
UserSpec(id,V) =
    let Aquire(s) =
            if s != <> then
                acquire(id,head(s)) -> Aquire(tail(s))
            else SKIP
        Release(s) =
            if s != <> then
                release(id,head(s)) -> Release(tail(s))
            else SKIP
        User(s) =
            Aquire(s);Release(s);User(s)
    within
        User(resources(id))
```

The behavioural restriction imposed by the resource allocation pattern is given by a conformance notion using the stable failure refinement relation `[F=`. The refinement relation ensures that user and resource atoms of the network meet their respective specification.

Definition 10 (Resource allocation behavioural restriction)**.** *Let $uset$ and $rset$ be the sets of users and resources atoms identifiers, respectively.*

$$\begin{aligned} BehaviourRA(V, uset, rset) \mathrel{\widehat{=}} \; & Behaviour(V, uset, \mathtt{UserSpec}, \mathtt{[F=}) \wedge \\ & Behaviour(V, rset, \mathtt{ResourceSpec}, \mathtt{[F=}) \end{aligned}$$

where: $Behaviour(V, S, Spec, R) = \forall\, id : S \bullet Spec(id, V)\; R\; Abs(id, V)$

Besides the behavioural restriction, this pattern also imposes a structural restriction, which is given by a conjunction of smaller conditions. The first condition, *partitions*, ensures that users and resources are two disjoint partitions of the network identifiers. The *disjointAlpha* condition guarantees that the alphabet of users and resources are disjoint, whereas *controlledAlpha* imposes that the shared events between users and resources must be the set of acquire and release events. Finally, *strictOrder* ensures that the transitive closure of the $>_{RA}$ relation, $>^{*}_{RA}$, is a strict total order.

Definition 11 (Resource allocation structural restriction)**.** *Let V be a network, users a set of user atom identifiers, resources a set of resource atom identifiers.*

$$
\begin{aligned}
&StructureRA(V, users, resources) \mathrel{\widehat{=}} \\
&\quad partitions(\operatorname{dom} V, users, resources) \wedge && (P)\\
&\quad disjointAlpha(V, resources) \wedge && (DARes)\\
&\quad disjointAlpha(V, users) \wedge && (DAUsers)\\
&\quad controlledAlpha(V, users, resources) \wedge && (CA)\\
&\quad strictOrder(>^{*}_{RA'}) && (SO)
\end{aligned}
$$

where:

- $partitions(S, P1, P2) \mathrel{\widehat{=}} S = P1 \cup P2 \wedge P1 \cap P2 = \emptyset$
- $disjointAlpha(V, S) \mathrel{\widehat{=}} \forall\, id_1, id_2 : S \bullet A(id_1, V) \cap A(id_2, V) = \emptyset$
- $controlledAlpha(V, S1, S2) \mathrel{\widehat{=}} \forall\, id_1 : S1, id_2 : S2 \bullet$
  $A(id_1, V) \cap A(id_2, V) = \{acquire(id_1, id_2), release(id_1, id_2)\}$
- $id_1 >_{RA} id_2 \mathrel{\widehat{=}} \exists\, id : users \bullet \exists\, i, j : \operatorname{dom} sequence(id) \bullet$
  $id_1 = sequence(id)(i) \wedge id_2 = sequence(id)(j) \wedge i < j$
- $id_1 >_{RA'} id_2 \mathrel{\widehat{=}} id_2 = id_1' \vee id >_{RA} id_2 \wedge id_1 = id'$

The compliance with the resource allocation pattern is given by the conformance to both behavioural and structural conformances; i.e. the network must satisfy both the *StructureRA* and *BehaviourRA* predicates.

As the purpose of the pattern is to avoid deadlock, we present a theorem which demonstrates that compliance to the resource allocation pattern prevents deadlock.

The resource allocation pattern guarantees that the resources have the *resourceProperty* and that the users atoms the *userProperty*, this guarantee is given by theorems 19 and 20.

Definition 12 (Resource property)**.** *Let id be an identifier of network V.*

$$
\begin{aligned}
&resourceProperty(id, V) \mathrel{\widehat{=}} \\
&\quad \forall\, f : Mfailures(Abs(id, V)) \bullet AcquiredResource(f, id, V) \vee ReleasedResource(f, id, V)
\end{aligned}
$$

where:

- $AcquiredResource(f, id, V) \mathrel{\widehat{=}}$
  $(odd(f.s) \wedge$
  $(\exists\, id_{u1} : users(id) \bullet odd(f.s \upharpoonright \{acquire(id_{u1}, id), release(id_{u1}, id)\}) \wedge$
  $\forall\, id_{u2} : users(id) \bullet id_{u1} \neq id_{u2} \Rightarrow$
  $even(f.s \upharpoonright \{acquire(id_{u1}, id), release(id_{u1}, id)\})))$

- $ReleasedResource(f, id, V) \mathrel{\widehat{=}}$
  $(even(f.s) \wedge (\forall\, id_u : users(id) \bullet acquire(id_u, id) \in (A(id, V) \setminus f.R) \wedge$
  $even(f.s \upharpoonright \{acquire(id_u, id), release(id_u, id)\})))$

Definition 13 (User property)**.** *Let id be an identifier of network V.*

$$
\begin{aligned}
&userProperty(id, V) \mathrel{\widehat{=}} \\
&\quad \forall\, f : Mfailures(Abs(id, V)) \bullet UserReleasing(f, id, V) \vee UserAcquiring(f, id, V)
\end{aligned}
$$

where:

- $UserReleasing(f, id, V) \mathrel{\widehat{=}}$
  $\exists\, id_r : resources \bullet$
  $((A(id, V) \setminus f.R) = \{release(id, id_r)\} \wedge$
  $odd(f.s \upharpoonright \{acquire(id, id_r), release(id, id_r)\}))$

- $UserAcquiring(f, id, V) \mathrel{\widehat{=}}$
  $\exists\, id_r : resources \bullet$
  $((A(id, V) \setminus f.R) = \{acquire(id, id_r)\} \wedge$
  $even(f.s \upharpoonright \{acquire(id, id_r), release(id, id_r)\}) \wedge$
  $min(r(f.s, id) \cup \{big\})) >_{RA} id_r$

- $r(s, id) \mathrel{\widehat{=}} \{id_r | id_r \in \operatorname{ran} resources(id) \wedge odd(s \upharpoonright \{acquire(id, id_r), release(id, id_r)\})\}$

The following theorem is the main result of this section it establishes that a network that is conform to the resource allocation pattern is deadlock free.

Theorem 8 (Deadlock free resource allocation network)**.** *Let V be a network users and resources two sets of identifiers.*

$$\textit{If } RA(V, users, resources) \textit{ then } V \textit{ is deadlock free.}$$

where: $RA(V, users, resources) \mathrel{\widehat{=}} StructureRA(V, users, resources) \wedge BehaviourRA(V, users, resources)$

Proof. By Theorem 12 and Theorem 2. □

4.1.2 Client/server pattern

The client/server pattern is used for architectures where an atom can behave as a server or as a client in the network. The events in the alphabets of atoms can be classified into client requests, server requests and responses. When the process offers a server request event it is in a server state, in which it has to offer all its server requests to its clients. This behaviour is described by the following specification. The specification allows the process to behave arbitrarily when performing non server request events; however if a server request is offered, it offers all server request events. The server request events of atom *id* is given by the function *serverRequests*(*id*).

Definition 14 (Behavioural server requests specification)**.** *Let id be an identifier of the atom in a network V and serverEvents a function that yield the set of server events of an atom given its identifier.*

```
ServerRequestsSpec(id,V) =
    let sEvs = serverRequests(id)
        othersEvs = diff(A(id,V),sEvs)
        Server = ((|~| ev : othersEvs @ ev -> SKIP)
            |~| ([] ev : sEvs @ ev -> SKIP)) ;
            Server
    within
        if not empty(othersEvs) then
            Server
        else
            RUN(sevs)
```

where:

```
RUN(evs) = [] ev : evs @ ev -> RUN(evs)
```

Also, as a server, if a request demands a response, it must offer one of the possible response events to that request. The function *responses* gives this set of the expected responses for a request event. The server must offer at least a response from this set. The `ServerResponsesSpec` specification describes this expected behaviour; after a server request event in *sEvs*, it must offer at least one of the *responses*(*req*) events.

Definition 15 (Behavioural server responses specification)**.** *Let id be an identifier of the atom being tested and serverEvents a function that yields the set of server events of an atom given its identifier.*

```
RequestsResponsesSpec(id,V) =
    let
        cEvs = clientRequests(id)
        sEvs = serverRequests(id)
        ClientRequestsResponsesSpec =
            (|~| ev : cEvs @ ev ->
```

```
                (if empty(responses(ev)) then SKIP
                else ([] res : responses(ev) @ res -> SKIP)))
        ServerRequestsResponsesSpec =
            (|~| ev : sEvs @ ev ->
                (if empty(responses(ev)) then SKIP
                else (|~| res : responses(ev) @ res -> SKIP)))
        C = ClientRequestsResponsesSpec;C
        S = ServerRequestsResponsesSpec;S
        CS = (ClientRequestsResponsesSpec
            |~| ServerRequestsResponsesSpec);CS
    within
        if empty(cEvs) and empty(sEvs) then STOP
        else
            if empty(cEvs) then S
            else
                if empty(sEvs) then C
                else CS
```

No restriction is imposed in client requests whatsoever; a process can always perform one of its client requests. Hence no specification of expected behaviour is required. On the other hand, the client must be able to accept any of the expected responses after performing a request. The client requests of an atom identified by *id* is given by the function *clientRequests*(*id*). This expected behaviour is given by the process `ClientResponsesSpec` where after an event from *clientsRequests*, the process offers all its response events, given by the same *responses* function.

The conformance relation of an atom's behaviour to the `ServerRequestsSpec` is given by the refinement relation in the stable revivals model. Both `ClientResponsesSpec` and `ServerResponsesSpec` conformance is ensured by the stable failure refinement relation.

Definition 16 (Client/server behavioural restriction)**.** *Let V be a network.*

$$\begin{aligned} BehaviourCS(V) \mathrel{\widehat=}\ & Behaviour(V, \operatorname{dom} V, \texttt{ServerRequestsSpec}, \texttt{[V=}) \wedge \\ & Behaviour(V, \operatorname{dom} V, \texttt{RequestResponsesSpec}, \texttt{[F=}) \end{aligned}$$

Similarly to the resource allocation structural restriction, the structural restriction of the client/server pattern is composed of a conjunction of smaller clauses. The *disjointAlpha* predicate ensures that the server events and client events of any atom are disjoint. The *controlledAlpha* predicate guarantees that the communication alphabet is restricted to client and server events. The *paired Events* guarantees that every server event has a client pair and vice-versa. Also, the *strictOrder* predicate guarantees that the transitive closure of the $>_{CS}$ relation, $(>^*_{CS})$, is a strict order.

Definition 17 (Client/server structural restriction)**.** *Let V be a network, $SRq(id) = serverRequests(id)$, $CRq(id) = clientRequests(id)$, $SRp(id) = \bigcup_{req \in SRq(id)} responses(req)$*

and $CRp(id) = \bigcup_{req \in CRq(id)} responses(req)$.

$$\begin{aligned} StructureCS(V) \mathrel{\widehat=}\ & disjointAlpha(\operatorname{dom} V) \land controlledAlpha(V, \operatorname{dom} V) \land \\ & pairedEvents(V, \operatorname{dom} V) \land strictOrder(>^*_{CS}) \end{aligned}$$

where:

- $disjointAlpha(S) \mathrel{\widehat=}$
  $\forall id : S \bullet SRq(id) \cap CRq(id) = \emptyset \land SRq(id) \cap CRp(id) = \emptyset \land$
  $SRp(id) \cap CRq(id) = \emptyset \land SRp(id) \cap CRp(id) = \emptyset$
- $controlledAlpha(V, S) \mathrel{\widehat=}$
  $\forall id : S \bullet AVoc(id, V) = SRq(id) \cup CRq(id) \cup SRp(id) \cup CRp(id)$
- $pairedRequests(V, S) \mathrel{\widehat=}$
  $\forall id : \operatorname{dom} V \bullet \forall req : SRq(id) \bullet \exists id' : \operatorname{dom} V \bullet req \in CRq(id') \land$
  $\forall id : \operatorname{dom} V \bullet \forall req : CRq(id) \bullet \exists id' : \operatorname{dom} V \bullet req \in SRq(id')$
- $id1 >_{CS} id2 \mathrel{\widehat=} CRq(id1) \cap SRq(id2) \neq \emptyset$

A network conforms to this predicate if the conjunction of the structural and behavioural restriction is satisfied.

In the same way as the one presented for the resource allocation pattern, we introduce a set of properties that a maximal failure of a atom must have if compliant to the Client/Server pattern. These properties are used as a specification of the maximal failures of the atoms in the proof of deadlock freedom.

Definition 18 (Client server property)**.**

$$\begin{aligned} & clientServerProperty(id, V) \mathrel{\widehat=} \\ & \quad \forall f : Mfailures(Abs(id, V)) \bullet \\ & \qquad ServerResponding(f, id, V) \lor ClientResponding(f, id, V) \lor \\ & \qquad ServerRequesting(f, id, V) \lor ClientRequesting(f, id, V) \end{aligned}$$

Definition 19 (ServerResponding predicate)**.**

$$\begin{aligned} & ServerResponding(f, id, V) \mathrel{\widehat=} \\ & \quad SResp(f, id, V) \land \exists ev : responses(last(f.s)) \bullet ev \in (A(id, V) \setminus f.R) \end{aligned}$$

where:
$SResp(f, id, V) \mathrel{\widehat=} f.s \neq \langle\rangle \land last(f.s) \in SRq(id) \land responses(last(f.s)) \neq \emptyset$

Definition 20 (ClientResponding predicate)**.**

$$\begin{aligned} & ClientResponding(f, id, V) \mathrel{\widehat=} \\ & CResp(f, id, V) \land (A(id, V) \setminus f.R) = responses(last(f.s)) \end{aligned}$$

where:
$CResp(f, id, V) \mathrel{\widehat=} f.s \neq \langle\rangle \land last(f.s) \in CRq(id) \land responses(last(f.s)) \neq \emptyset$

Definition 21 (ServerResquesting predicate)**.**

$ServerResquesting(f, id, V) \mathrel{\widehat{=}}$
$\quad SReq(f, id, V) \wedge SRq(id) \subseteq (A(id, V) \setminus f.R)$

where:
$SReq(f, id, V) \mathrel{\widehat{=}} (f.s = \langle\rangle \vee last(f.s) \in responses(id) \vee last(f.s) \in requests(id) \wedge responses(last(f.s)) = \emptyset) \wedge SRq(id) \not\subseteq f.R$

Definition 22 (ClientResquesting predicate)**.**

$ClientResquesting(f, id, V) \mathrel{\widehat{=}}$
$\quad CReq(f, id, V) \wedge \exists\, req : CRq(id) \bullet req \in (A(id, V) \setminus f.R)$

where: $CReq(f, id, V) \mathrel{\widehat{=}} (f.s = \langle\rangle \vee last(f.s) \in responses(id) \vee last(f.s) \in requests(id) \wedge responses(last(f.s)) = \emptyset) \wedge f.R \cap SRq(id) = \emptyset$

The goal of preventing deadlock is achieved by this pattern as stated by the following theorem.

Theorem 9 (Network CS conform is deadlock free)**.** *Let V be a network.*

If $ConformCS(V)$ then V is deadlock free.

where: $ConformCS(V) = BehaviourCS(V) \wedge StructureCS(V)$

Proof. We conduct the proof by assuming $ConformCS(V)$ and proving that in this conditions V is deadlock free.

$\Longrightarrow$ [Assumption]
$state(\sigma) \wedge max(\sigma, V)$

$\Longrightarrow$ [Theorem 34.]
$\forall\, id : \operatorname{dom} V \bullet clientServerProperty(id)$

$\Longrightarrow$ [$clientServerProperty(id)$ def]
$\forall\, id : \operatorname{dom} V \bullet ServerResponding(f, id, V) \vee ClientResponding(f, id, V) \vee ServerRequesting(f, id, V) \vee ClientRequesting(f, id, V)$

Here, we split the proof into 5 cases.

- $\exists\, id : \operatorname{dom} V \bullet ClientResponding(f, id, V)$
- $\exists\, id : \operatorname{dom} V \bullet ServerResponding(f, id, V)$
- $\exists\, C \bullet Cycle(C, \sigma) \wedge \exists\, i, i' : \operatorname{dom} C \bullet ClientRequesting(f(C(i)), C(i), V) \wedge ServerRequesting(f(C(i')), C(i'), V)$
- $\exists\, C \bullet Cycle(C, \sigma) \wedge \forall\, i : \operatorname{dom} C \bullet ClientResquesting(f(C(i)), i, V)$
- $\exists\, C \bullet Cycle(C, \sigma) \wedge \forall\, i : \operatorname{dom} C \bullet ServerResquesting(f, id, V)$

Case 1. *Let $f = \rho(\sigma, id, V)$, we prove for Case 1 ($\exists\, id : \operatorname{dom} V \bullet ClientResponding(f, id, V)$) a deadlock cannot occur. We start by assuming that there is a client responding process as denoted by predicate $ClientReponding$.*

$$\exists\, id : \operatorname{dom} V \bullet ClientResponding(f, id, V)$$

$$\Longrightarrow \qquad \textit{[ClientResponding(f, id, V) def]}$$

$$\exists\, id : \operatorname{dom} V \bullet last(f.s) \in CReq(id) \wedge responses(last(f.s)) \neq \emptyset$$

$$\Longrightarrow \qquad \textit{[pairedEvents def]}$$

$$(\exists\, id : \operatorname{dom} V \bullet last(f.s) \in CReq(id) \wedge responses(last(f.s)) \neq \emptyset) \wedge$$
$$(\exists\, id' : \operatorname{dom} V \bullet last(f.s) \in SReq(id') \wedge responses(last(f.s)) \neq \emptyset)$$

Since after two processes agreeing on a request event, they must agree on a response event, since process id has not performed any event after $last(f.s)$, then process id' can not have performed any event either, hence $last(f.s) = last(f'.s)$.

$$\Longrightarrow \qquad [last(f.s) = last(f'.s)]$$

$$(\exists\, id : \operatorname{dom} V \bullet last(f.s) \in CReq(id) \wedge responses(last(f.s)) \neq \emptyset) \wedge$$
$$(\exists\, id' : \operatorname{dom} V \bullet last(f'.s) \in SReq(id') \wedge responses(last(f'.s)) \neq \emptyset)$$

$$\Longrightarrow \qquad \textit{[ClientResponding(f, id, V) and ServerResponding(f'id', V) hold]}$$

$$(\exists\, id : \operatorname{dom} V \bullet (A(id, V) \setminus f.R) = responses(last(f.s))) \wedge$$
$$(\exists\, id' : \operatorname{dom} V \bullet \exists\, ev : responses(last(f'.s)) \bullet ev \in (A(id', V) \setminus f'.R))$$

$$\Longrightarrow \qquad \textit{[ST and PC]}$$

$$\exists\, id, id' : \operatorname{dom} V \bullet \exists\, ev : responses(last(f'.s)) \bullet ev \notin (A(id', V) \cap f'.R) \cup (A(id, V) \cap f.R)$$

By triple disjointness ev cannot belong to any alphabet other than alphabets $A(id', V)$ and $A(id, V)$. Hence, $ev \notin refusals(\sigma)$.

$$\Longrightarrow$$

$$\exists\, id, id' : \operatorname{dom} V \bullet \exists\, ev : responses(last(f'.s)) \bullet ev \notin refusals(\sigma)$$

$$\Longrightarrow \qquad [ev \in \Sigma]$$

$$\exists\, id, id' : \operatorname{dom} V \bullet \exists\, ev : responses(last(f'.s)) \bullet refusals(\sigma) \neq \Sigma_V$$

$$\Longrightarrow \qquad \textit{[PC]}$$

$$refusals(\sigma) \neq \Sigma$$

Case 2. *Regarding this case we can prove it in a very similar reasoning to the case 1. We assume that there is an $ServerResponding$ atom in the network and we show that there is a corresponding $ClientResponding$. Hence, we prove that they agree on an response, what proves that in this state both processes can perform an event making this state not deadlocked.*

Case 3. *Let V be an arbitrary network, σ an arbitrary state, $f(id) = \rho(\sigma, C(id), V)$, we want to prove that if $\exists\, i, j' \bullet ClientRequesting(f(C(i)), C(i), V) \wedge ServerRequesting(f(i'), C(i'), V)$ then the state is deadlock free.*

First of all, let us assume that there is a cycle of ungranted requests in this state. A cycle is a pair (C, σ) where C is a sequence of identifiers of the network V and σ a state of this network.

If there is a ServerResponding or a ClientResponding atom in the cycle, this implies that there is such a atom in the network and by the two previous already demonstrated cases we conclude that the network is deadlock free. Hence, we only consider cycles without process behaving according to these predicates. Hence, processes can behave according to either as ClientRequesting or as ServerRequesting. Therefore, for our case, the following predicate holds.

- $\forall i : \operatorname{dom} C \bullet \mathit{ClientRequesting}(f(C(i)), C(i), V) \vee \mathit{ServerRequesting}(f(C(i)), C(i), V)$

$\exists C \bullet \mathit{Cycle}(C, \sigma) \wedge \exists i, i' : \operatorname{dom} C \bullet$ *[Assumption 1]*
$\mathit{ClientRequesting}(f(C(i)), C(i), V) \wedge \mathit{ServerRequesting}(f(C(i')), C(i'), V)$

$\Longrightarrow$ *[Lemma 21]*
$\exists C \bullet \mathit{Cycle}(C, \sigma) \wedge \exists i : \operatorname{dom} C \bullet$
$\mathit{ClientRequesting}(f(C(i)), C(i), V) \wedge \mathit{ServerRequesting}(f(C(i \oplus 1)), C(i \oplus 1), V)$

$\Longrightarrow$ *[Cycle def]*
$\exists i : \operatorname{dom} C \bullet$
$\mathit{ClientRequesting}(f(C(i)), C(i), V) \wedge \mathit{ServerRequesting}(f(C(i \oplus 1)), C(i \oplus 1), V) \wedge$
$\mathit{ungranted\_request}(\sigma, C(i), C(i \oplus 1), V)$

$\Longrightarrow$ *[ClientRequesting def]*
$\exists i : \operatorname{dom} C \bullet$
$\exists \mathit{req} : \mathit{CRq}(C(i)) \bullet \mathit{req} \in (A(C(i), V) \setminus f.R) \wedge$
$\mathit{ServerRequesting}(f(C(i \oplus 1)), C(i \oplus 1), V) \wedge$
$\mathit{ungranted\_request}(\sigma, C(i), C(i \oplus 1), V)$

$\Longrightarrow$ *[ServerRequesting def]*
$\exists i : \operatorname{dom} C \bullet$
$\exists \mathit{req} : \mathit{CRq}(C(i)) \bullet \mathit{req} \in (A(C(i), V) \setminus f(C(i)).R) \wedge$
$\mathit{SRq}(C(i \oplus 1)) \subseteq (A(C(i \oplus 1), V) \setminus f(C(i \oplus 1)).R) \wedge$
$\mathit{ungranted\_request}(\sigma, C(i), C(i \oplus 1), V)$

$\Longrightarrow$ *[pairedEvents definition]*
$\exists i : \operatorname{dom} C \bullet$
$\exists \mathit{req} : \mathit{CRq}(C(i)) \bullet \mathit{req} \in (A(C(i), V) \setminus f(C(i)).R) \wedge$
$\mathit{SRq}(C(i \oplus 1)) \subseteq (A(C(i \oplus 1), V) \setminus f(C(i \oplus 1)).R) \wedge$
$\mathit{ungranted\_request}(\sigma, C(i), C(i \oplus 1), V) \wedge$
$\exists \mathit{id} : \operatorname{dom} V \bullet \mathit{req} \in \mathit{SRq}(\mathit{id})$

Here we split into two cases:

- $\mathit{id} = C(i \oplus 1)$
- $\mathit{id} \neq C(i \oplus 1)$

Case 3.1 ($id = C(i \oplus 1)$).

$$
\begin{array}{lr}
\Longrightarrow & [id = C(i \oplus 1)] \\
\exists\, i : \operatorname{dom} C \bullet & \\
\exists\, req : CRq(C(i)) \bullet req \in (A(C(i), V) \setminus f(C(i)).R) \wedge & \\
SRq(C(i \oplus 1)) \subseteq (A(C(i \oplus 1), V) \setminus f(C(i \oplus 1)).R) \wedge & \\
ungranted\_request(\sigma, C(i), C(i \oplus 1), V) \wedge & \\
req \in SRq(C(i \oplus 1)) & \\
\Longrightarrow & [ST \text{ and } PC] \\
\exists\, i : \operatorname{dom} C \bullet & \\
\exists\, req : CRq(C(i)) \bullet & \\
req \in ((A(C(i \oplus 1), V) \setminus f(C(i \oplus 1)).R) \cap (A(C(i \oplus 1), V) \setminus f(C(i \oplus 1)).R)) \wedge & \\
ungranted\_request(\sigma, C(i), C(i \oplus 1), V) & \\
\Longrightarrow & [ST \text{ and } PC] \\
\exists\, i : \operatorname{dom} C \bullet & \\
(A(C(i \oplus 1), V) \setminus f(C(i \oplus 1)).R) \cap (A(C(i \oplus 1), V) \setminus f(C(i \oplus 1)).R) \neq \emptyset \wedge & \\
ungranted\_request(\sigma, C(i), C(i \oplus 1), V) & \\
\Longrightarrow & [ungrantedness \text{ def}] \\
\exists\, i : \operatorname{dom} C \bullet & \\
\neg ungrantedness(\sigma, C(i), C(i \oplus 1), V) \wedge & \\
ungranted\_request(\sigma, C(i), C(i \oplus 1), V) & \\
\Longrightarrow & [ungranted\_request \text{ def and } PC] \\
false & \\
\Longrightarrow & [ungranted\_request \text{ def and } PC] \\
(\exists\, i, i' : \operatorname{dom} C \bullet & \\
ClientRequesting(f(C(i)), C(i), V) \wedge ServerRequesting(f(C(i')), C(i'), V)) & \\
\Rightarrow \forall\, C \bullet \neg Cycle(C, \sigma) & \\
\Longrightarrow & [\text{Theorem 1}] \\
refusals(\sigma) \neq \Sigma &
\end{array}
$$

Case 3.2 ($id \neq C(i \oplus 1)$).

$$
\begin{array}{lr}
\Longrightarrow & [id \neq C(i \oplus 1)] \\
\exists\, i : \operatorname{dom} C \bullet & \\
\exists\, req : CRq(C(i)) \bullet req \in (A(C(i), V) \setminus f(C(i)).R) \wedge & \\
SRq(C(i \oplus 1)) \subseteq (A(C(i \oplus 1), V) \setminus f(C(i \oplus 1)).R) \wedge & \\
ungranted\_request(\sigma, C(i), C(i \oplus 1), V) \wedge & \\
req \notin SRq(C(i \oplus 1)) & \\
\Longrightarrow & [ST \text{ and } PC] \\
\exists\, i : \operatorname{dom} C \bullet & \\
(A(C(i), V) \setminus f(C(i)).R) \cap A(C(i \oplus 1), V) = \emptyset \wedge & \\
ungranted\_request(\sigma, C(i), C(i \oplus 1), V) &
\end{array}
$$

$\Longrightarrow$ *[request def and PC]*
$\exists\, i : \mathrm{dom}\, C \bullet$
$\neg request(\sigma, C(i), C(i \oplus 1), V) \wedge$
$ungranted\_request(\sigma, C(i), C(i \oplus 1), V)$

$\Longrightarrow$ *[ungranted_request def and PC]*
$false$

$\Longrightarrow$ *[ungranted_request def and PC]*
$(\exists\, i, i' : \mathrm{dom}\, C \bullet$
$ClientRequesting(f(C(i)), C(i), V) \wedge ServerRequesting(f(C(i')), C(i'), V))$
$\Rightarrow (\forall\, C \bullet \neg Cycle(C, \sigma))$

$\Longrightarrow$ *[Theorem 1]*
$refusals(\sigma) \neq \Sigma$

Case 4 $(Cycle(C, \sigma) \wedge \forall\, i : \mathrm{dom}\, C \bullet ClientRequesting(f(C(i)), C(i), V))$**.** *For this case, we prove that a cycle cannot happen since the strict order* $(\mathrm{dom}\, V, >^*_{CS}$ $)$ *prevents it.*

$Cycle(C, \sigma) \wedge \forall\, i : \mathrm{dom}\, C \bullet ClientRequesting(f(C(i)), C(i), V)$

$\Longrightarrow$ *[*$CRq(i) \cap SRq(i \oplus 1) \neq \emptyset$*]*
$\forall\, i : \mathrm{dom}\, C \bullet C(i) >_{CS} C(i \oplus 1)$

$\Longrightarrow$ *[transitivity of* $>^*_{CS}$*]*
$\forall\, i : \mathrm{dom}\, C \bullet C(i) >^*_{CS} C(i)$

$\Longrightarrow$ *[*$strictOrder(\mathrm{dom}\, V, >^*_{CS})$*]*
$\forall\, i : \mathrm{dom}\, C \bullet C(i) >^*_{CS} C(i) \wedge irreflexive(\mathrm{dom}\, V, >^*_{CS})$

$\Longrightarrow$ *[irreflexive def and PC]*
$false$

$\Longrightarrow$ *[irreflexive def and PC]*
$(\forall\, i : \mathrm{dom}\, C \bullet ClientRequesting(f(C(i)), C(i), V)) \Rightarrow (\forall\, C \bullet \neg Cycle(C, \sigma))$

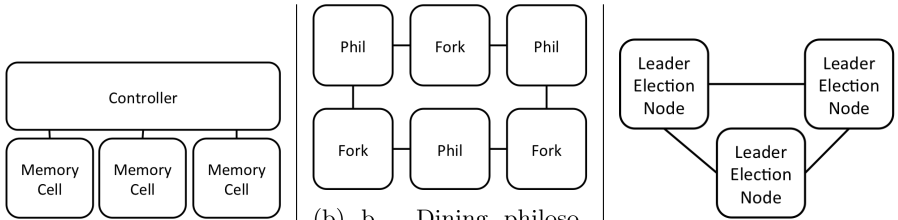
$\Longrightarrow$ *[Theorem 1]*
$refusals(\sigma) \neq \Sigma$

Case 5 $(Cycle(C, \sigma) \wedge \forall\, i : \mathrm{dom}\, C \bullet ServerRequesting(f(C(i)), C(i), V))$**.** *For this case, we use the same reasoning as the one used in the last case but instead of using the relation* $>_{CS}$*, we use its dual,* $<_{CS}$

$\square$

5 Experimental analysis

As a proof of concept of our strategy, we have applied the formalised patterns and conflict freedom assertion to verify deadlock freedom for three examples: a ring buffer, the asymmetric dining philosophers and a leadership election algorithm. The CSP models of all the three examples are parametrised to allow



(a) a. Ring buffer

(b) b. Dining philosophers

(c) c. Leadership election

Figure 1: Communication architecture with N = 3

instances with different number of processes. The CSP models can be found in Appendix D.

The ring buffer stores data in a circular way. This system is composed of a controller which is responsible for inputting and outputting data, and a set of memory cells to store data. The controller is responsible for storing input data in the appropriate cell according to its information about the top and bottom indices of the buffer. It also possesses a cache cell where it stores the data ready to be read. This system has an acyclic topology as it can be seen as a tree where the controller is the root and the memory cells its leaves. We parametrised this model by N, the number of cells to store data. Its communication architecture for a model with $N = 3$ is depicted in Figure 1a.

The dining philosophers is a classical example that consists of philosophers that try to acquire forks in order to eat. It is a classical deadlock problem and its asymmetric version obeys our resource allocation pattern restrictions. The forks are the resources and the philosophers the users. In the asymmetric case, every philosopher acquires its left fork, then its right one, but one has an asymmetric behaviour acquiring first the right and then the left fork. This is a cyclic network that has a ring topology, and a classical example of the resource allocation pattern. This model is parametrised by N the number of philosophers. Its communication architecture for a model with $N = 3$ is depicted in Figure 1b.

The last example is a simplified model of a distributed synchronised leadership election system. The nodes are composed of a controller, a memory, a receiver and a transmitter and they exchange data to elect the leader of the network. Every node can communicate with every other node, hence we have a cyclic fully connected graph. For this model we applied the client/server pattern as this leadership election model conforms to this pattern. We parametrised this model by N the number of leadership election nodes. Its communication architecture for a model with $N = 3$ is depicted in Figure 1c.

In order to demonstrate, in practice, that local analysis avoids combinatorial explosion, we have conducted a comparative analysis of two verification approaches for the three examples, all using the FDR tool: (i) analysis of the complete model; (ii) local analysis of the model using the refinement assertions presented in Section 4. For the analysis of our strategy (ii), we only assess the time for verifying behavioural constraints. Since the structural restrictions

	Ring Buffer			Dining Philosophers			Leader Election		
N	#Procs	(i)	(ii)	#Procs	(i)	(ii)	#Procs	(i)	(ii)
3	4	0.02	0.01	6	0.19	0.09	12	*	8.67
5	6	0.161	0.535	10	0.109	0.21	20	*	18
10	11	86.79	3.12	20	701.05	0.4	40	*	62
20	21	*	21.92	40	*	1	80	*	442
30	31	*	85.35	60	*	2.28	120	*	1926

$*$ Exceed the execution limit of 1 hour

Table 1: Performance comparison measured in seconds.

can be static analysed, they represent a negligible value if compared to the behavioural constraints.

We conducted the analysis for different instances of N's (3, 5, 10, 20, 30), as explained before; these are summarised in Table 1. In the table we present the amount of time involved in each case. We used a dedicated server with an 8 core Intel(R) Xeon(R) 2.67GHz and 16 GB of RAM in an Ubuntu 4.4.3 system.

The results demonstrate how the time for deadlock verification can grow exponentially with the linear increase of the number of processes for global methods such as (i). Also, it demonstrates that our approach, based on patterns that support local analysis, seems promising; to our knowledge, it is the first sound and be the only automated strategy for guaranteeing deadlock freedom for complex systems. Notice, particularly, that our strategy (ii) allows one to verify a leadership election system with 30 nodes in less than 35 minutes, a very promising result in dealing with a complex system involving a fully connected graph of components. On the other hand, global analysis of the complete model in FDR is unable to give an answer in the established time limit for a 3 node instance. In order to give an idea of the size of this system with 30 nodes, the processes controller, receiver, transmitter and memory have 854, 271, 263 and 99 states, respectively. This means that the leader election system can have up to $854^{30} \times 271^{30} \times 263^{30} \times 99^{30}$ states. Another consideration is that local analysis also enables the use of parallel cores to verify simultaneously different processes, which would reduce the amount of time for verification even further.

6 Conclusion and related work

Our verification strategy focuses on a local analysis of deadlock freedom of design models of concurrent systems which obey certain architectural patterns. Although this method is not complete, it already covers a vast spectrum of systems, those that are conflict free systems, as well as cyclic systems that can be designed in terms of the formalised patterns. The strategy seems promising in terms of performance, applicability and complexity mastering, as evidenced by the application of the strategy for complex systems such as a distributed leadership election example.

Roscoe and Brookes developed a structured model for analysing deadlock in networks [4]. They created the model based on networks of processes and a body of concepts that helped to analyse networks in a more elegant and abstract way. Roscoe and Dathi also contributed by developing a proof method for deadlock freedom [10]. They have built a method to prove deadlock freedom based on variants, similar to the ones used to prove loop termination. In their work, they also start to analyse some of the patterns that arise in deadlock free systems. Although their results enable one to verify locally a class of networks, there is no framework available that implements their results such as the one presented here. A more recent work by Roscoe et al. [11] presents some compression techniques, which are able to check the dining philosopher example for 10^{100} processes. Compression techniques are an important complementary step for further improving our strategy.

Following these initial works, Martin defined some design rules to avoid deadlock freedom [6]. He also developed an algorithm and a tool with the specific purpose of deadlock verification, the Deadlock checker [7], which reduces the problem of deadlock checking to the quest of cycles of ungranted requests, in live networks. The algorithm used by this tool can also incur an exponential explosion in the state space to be verified, as the quest of a cycle of ungranted request can be as hard as the quest of finding a deadlocked state.

In a recent work, Ramos et al. developed a strategy to compose systems guaranteeing deadlock freedom for each composition [8]. The main drawback with their method is the lack of compositional support to cyclic networks. One of the rules presented there is able to, in a compositional way, connect components in order to build a tree topology component. They presented a rule to deal with cyclic components but it is not compositional, in the sense that the verification of its proviso is not local, i.e. it must be performed in the entire system. Our strategy complements and can be easily combined with this compositional approach. A distinguishing feature of our strategy is precisely the possibility of combining it with other systematic approaches to analysis.

As future work we plan to formalise additional patterns, such as the cyclic communicating pattern. Also, we plan to carry out further practical experiments and implement an elaborate framework to support the entire strategy, running FDR in background to carry out the analyses.

A General theorems

Theorem 10 (Maximal failures induced by refinement)**.** *Let P and Q be two arbitrary processes.*

$$P \sqsubseteq_F Q \Rightarrow Mfailures(Q) \subseteq \{f : failures(P) | f.R \supseteq \overline{initials(P/f.s)}^{\Sigma}\}$$

Proof. The proof is conducted by contradiction.

$$P \sqsubseteq_F Q \wedge Mfailures(Q) \not\subseteq \{f | f : failures(P) \wedge f.R \supseteq \overline{initials(P/f.s)}^{\Sigma}\} \quad \text{[Assumption]}$$

$\Longrightarrow$ [$\not\subseteq$ def]

$P \sqsubseteq_F Q \wedge \exists mf : Mfailures(Q) \bullet mf \notin \{f | f : failures(P) \wedge f.R \supseteq \overline{initials(P/f.s)}^{\Sigma}\}$

$\Longrightarrow$ [$\not\subseteq$ def]

$P \sqsubseteq_F Q \wedge \exists mf : Mfailures(Q) \bullet mf \notin failures(P) \vee mf.R \not\supseteq \overline{initials(P/mf.s)}^{\Sigma}\}$

Here we have to prove the contradiction for two cases:

- $mf \notin failures(P)$
- $mf.R \supseteq \overline{initials(P/mf.s)}^{\Sigma}$

Case 1. $mf \notin failures(P)$

$\Longrightarrow$ *[$mf \notin failures(P)$ holds]*

$P \sqsubseteq_F Q \wedge \exists mf : Mfailures(Q) \bullet mf \notin failures(P)$

$\Longrightarrow$ *[$Mfailures$ def]*

$P \sqsubseteq_F Q \wedge \exists mf : failures(Q) \bullet mf \notin failures(P)$

$\Longrightarrow$ *[$\not\subseteq$ def]*

$P \sqsubseteq_F Q \wedge failures(Q) \not\subseteq failures(P)$

$\Longrightarrow$ *[$\sqsubseteq_F$ def and PC]*

$false$

Case 2. $mf.R \supseteq \overline{initials(P/mf.s)}^{\Sigma}$

$\Longrightarrow$ *[$mf.R \supseteq \overline{initials(P/mf.s)}^{\Sigma}$ holds]*

$P \sqsubseteq_F Q \wedge$

$\exists mf : Mfailures(Q) \bullet mf.R \not\supseteq \overline{initials(P/mf.s)}^{\Sigma}$

$\Longrightarrow$ *[$\not\supseteq$ def]*

$P \sqsubseteq_F Q \wedge$

$\exists mf : Mfailures(Q) \bullet \exists ev : \overline{initials(P/mf.s)}^{\Sigma} \bullet ev \notin mf.R$

$\Longrightarrow$ *[$\overline{initials(P/mf.s)}^{\Sigma}$ def]*

$P \sqsubseteq_F Q \wedge$

$\exists mf : Mfailures(Q) \bullet \exists ev : \Sigma \bullet ev \notin initials(P/mf.s) \wedge ev \notin mf.R$

$\Longrightarrow$ *[Healthiness F2]*

$P \sqsubseteq_F Q \wedge$

$\exists mf : Mfailures(Q) \bullet \exists ev : \Sigma \bullet ev \notin initials(P/mf.s) \wedge ev \in initials(Q/mf.s)$

$\Longrightarrow$ *[Healthiness T2]*

$P \sqsubseteq_F Q \wedge$

$\exists (mf : Mfailures(Q) \bullet$

$(\exists ev : \Sigma \bullet ev \notin initials(P/mf.s) \wedge ev \in initials(Q/mf.s)) \wedge mf.s \in traces(Q))$

Case 2.1. $mf.s \in traces(P)$

$$\Longrightarrow \qquad [mf.s \in traces(P) \text{ holds}]$$
$$P \sqsubseteq_F Q \land (\exists\, mf : Mfailures(Q) \bullet$$
$$(\exists\, ev : \Sigma \bullet ev \notin initials(P/mf.s) \land ev \in initials(Q/mf.s))$$
$$\land\, mf.s \in traces(Q) \land mf.s \in traces(P))$$

$$\Longrightarrow \qquad [initials \text{ def}]$$
$$P \sqsubseteq_F Q \land (\exists\, mf : Mfailures(Q) \bullet$$
$$\exists\, ev : \Sigma \bullet mf.s\hat{\ }\langle ev\rangle \notin traces(P) \land mf.s\hat{\ }\langle ev\rangle \in traces(Q))$$

$$\Longrightarrow \qquad [PC]$$
$$P \sqsubseteq_F Q \land (\exists\, mf : Mfailures(Q) \bullet$$
$$\exists\, ev : \Sigma \bullet \exists\, s \bullet s \notin traces(P) \land s \in traces(Q))$$

$$\Longrightarrow \qquad [PC]$$
$$P \sqsubseteq_F Q \land \exists\, s \bullet s \notin traces(P) \land s \in traces(Q)$$

$$\Longrightarrow \qquad [\not\supseteq \text{ def}]$$
$$P \sqsubseteq_F Q \land traces(P) \not\supseteq traces(Q)$$

$$\Longrightarrow \qquad [\sqsubseteq_F \text{ def and } PC]$$
$$false$$

Case 2.2. $mf.s \notin traces(P)$

$$\Longrightarrow \qquad [mf.s \notin traces(P) \text{ holds}]$$
$$P \sqsubseteq_F Q \land (\exists\, mf : Mfailures(Q) \bullet$$
$$(\exists\, ev : \Sigma \bullet ev \notin initials(P/mf.s) \land ev \in initials(Q/mf.s))$$
$$\land\, mf.s \in traces(Q) \land mf.s \notin traces(P))$$

$$\Longrightarrow \qquad [mf.s \notin traces(P)]$$
$$P \sqsubseteq_F Q \land (\exists\, mf : Mfailures(Q) \bullet mf.s \in traces(Q) \land mf.s \notin traces(P))$$

$$\Longrightarrow \qquad [PC]$$
$$P \sqsubseteq_F Q \land (\exists\, mf : Mfailures(Q) \bullet \exists\, s \bullet s \in traces(Q) \land s \notin traces(P))$$

$$\Longrightarrow \qquad [PC]$$
$$P \sqsubseteq_F Q \land (\exists\, s \bullet s \in traces(Q) \land s \notin traces(P))$$

$$\Longrightarrow \qquad [\not\supseteq \text{ def}]$$
$$P \sqsubseteq_F Q \land traces(Q) \not\supseteq traces(P)$$

$$\Longrightarrow \qquad [\sqsubseteq_F \text{ and } PC]$$
$$false$$

$\square$

Theorem 11 (Maximal revivals induced by stable revival refinement)**.** *Let P and Q be two deadlock free processes.*

$P \sqsubseteq_V Q \Rightarrow Mrevivals(Q) \subseteq MCrevivals(P)$

where: $Mrevivals(Q) \mathrel{\widehat=} \{r | r \in revivals(Q) \land max(r, Q)\}$

$MCrevivals(Q) \mathrel{\widehat=} \{r | r \in revivals(Q) \land r.R \supseteq \overline{initials(failure(r))}^{\Sigma} \land r.R \supseteq \overline{initials(r.s)}^{\Sigma}\}$

Proof.

$$P \sqsubseteq_{\mathtt{V}} Q \wedge Mrevivals(Q) \not\subseteq MCrevivals(P) Assumption$$
$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge \exists r : Mrevivals(Q) \bullet r \notin MCrevivals(P)$$
$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge \exists r : Mrevivals(Q) \bullet r \notin revivals(P) \vee$$
$$r.R \not\supseteq \overline{initials(P/failure(r))}^{\Sigma} \vee$$
$$r.R \supseteq \overline{initials(r.s)}^{\Sigma}$$

Case 1.

$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge \exists r : Mrevivals(Q) \bullet r.R \not\supseteq \overline{initials(P/failure(r))}^{\Sigma}$$
$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge \exists r : Mrevivals(Q) \bullet \exists ev : \overline{initials(P/failure(r))}^{\Sigma} \bullet ev \notin r.R$$
$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet ev \notin initials(P/failure(r)) \wedge ev \notin r.R$$
$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet ev \notin initials(P/failure(r)) \wedge ev \in initials(Q/failure(r))$$
$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet (r.s, r.R, ev) \notin revivals(P) \wedge (rs.r.R, ev) \in revivals(Q)$$
$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge \exists r : Mrevivals(Q) \bullet \exists r : revivals(Q) \bullet r \notin revivals(P)$$
$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge \exists r : Mrevivals(Q) \bullet revivals(Q) \not\subseteq revivals(P)$$
$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge revivals(Q) \not\subseteq revivals(P)$$
$$\implies$$
$$false$$

Case 2.

$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge \exists r : Mrevivals(Q) \bullet r.R \not\supseteq \overline{initials(P/r.s)}^{\Sigma}\}$$
$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge \exists r : Mrevivals(Q) \bullet \exists ev : \overline{initials(P/r.s)}^{\Sigma}\} \bullet ev \notin r.R$$
$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet ev \notin initials(P/r.s) \bullet ev \notin r.R$$

Case 2.1 ($ev \notin initials(Q/r.s)$)**.**

$$\implies$$
$$P \sqsubseteq_{\mathtt{V}} Q \wedge \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet ev \notin initials(P/r.s) \bullet ev \notin r.R \wedge ev \notin initials(Q/r.s)$$

$\Longrightarrow$
$P \sqsubseteq_{V=} Q \land \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet ev \notin initials(P/r.s) \bullet ev \notin r.R \land$
$\forall r' : revivals(Q) \bullet r'.a \neq ev$

$\Longrightarrow$
$P \sqsubseteq_{V=} Q \land \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet ev \notin initials(P/r.s) \bullet ev \notin r.R \land$
$\forall r' : revivals(Q) \bullet failure(r') = failure(r) \Rightarrow r'.a \neq ev$

$\Longrightarrow$
$P \sqsubseteq_{V=} Q \land \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet ev \notin initials(P/r.s) \bullet ev \notin r.R \land$
$(r.s, r.R \cup ev, r.a) \in revivals(Q)$

$\Longrightarrow$
$P \sqsubseteq_{V=} Q \land \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet ev \notin initials(P/r.s) \bullet ev \notin r.R \land$
$(r.s, r.R \cup ev, r.a) \in revivals(Q) \land r \subset (r.s, r.R \cup ev, r.a)$

$\Longrightarrow$
$P \sqsubseteq_{V=} Q \land \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet ev \notin initials(P/r.s) \bullet ev \notin r.R \land$
$(r.s, r.R \cup ev, r.a) \in revivals(Q) \land \neg max(r, Q)$

$\Longrightarrow$
$false$

Case 2.2 ($ev \in initials(Q/r.s)$)**.**

$\Longrightarrow$
$P \sqsubseteq_{V=} Q \land \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet ev \notin initials(P/r.s) \bullet ev \notin r.R \land ev \in initials(Q/r.s)$

$\Longrightarrow$
$P \sqsubseteq_{V=} Q \land \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet r.s\hat{}\langle ev\rangle \notin traces(P) \land r.s\hat{}\langle ev\rangle \in traces(Q) \land$

$\Longrightarrow$
$P \sqsubseteq_{V=} Q \land \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet \exists s : traces(Q) \bullet s \notin traces(P)$

$\Longrightarrow$
$P \sqsubseteq_{V=} Q \land \exists r : Mrevivals(Q) \bullet \exists ev : \Sigma \bullet traces(Q) \not\subseteq traces(P)$

$\Longrightarrow$ *[PC]*
$P \sqsubseteq_{V=} Q \land traces(Q) \not\subseteq traces(P)$

$\Longrightarrow$ *[PC and $\sqsubseteq_{V=}$ def]*
$false$

Case 3. *The case where $\exists r : Mrevivals(Q) \bullet r \notin refusals(P)$ is trivial.*

□

B Resource allocation auxiliary lemmas

Theorem 12 (RA conformance imply ungranted requests strict order)**.** *Let V be a network, users and resources two partitions of this network, and id_1 and id_2 two identifiers of this network. Assuming $RA(V, users, resources)$:*

$$\forall \sigma ; id_1, id_2 : \operatorname{dom} V \bullet state(\sigma, V) \wedge max(\sigma, V) \wedge id_1 \neq id_2 \wedge$$
$$ungranted\_request(\sigma, id_1, id_2, V) \Rightarrow g(\sigma, id_1) >^*_{RA'} g(\sigma, id_2)$$

where:

- $g(\sigma, id) \mathrel{\widehat{=}}$

$$\begin{array}{ll} r(\rho(\sigma, id, V).s)' & id \in users \\ big & id \in resources \wedge even(\rho(\sigma, id, V).s) \\ id & id \in resources \wedge odd(\rho(\sigma, id, V).s) \end{array}$$

Proof. Let V be an arbitrary network, id_1 and id_2 two identifiers of this network, σ an arbitrary state of this network, and $f_1 = \rho(\sigma, id_1)$ and $f_2 = \rho(\sigma, id_2)$.

$$id_1 \in \operatorname{dom} V \wedge id_2 \in \operatorname{dom} V \wedge state(\sigma, V) \wedge max(\sigma, V) \wedge \quad \text{[Assumption]}$$
$$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V)$$

Since $Paritions$ holds, there are 4 cases for combinations of id_1 and id_2:

- Case 1: $id_1 \in resources \wedge id_2 \in users$
- Case 2: $id_1 \in resources \wedge id_2 \in resources$
- Case 3: $id_1 \in users \wedge id_2 \in resources$
- Case 4: $id_1 \in users \wedge id_2 \in users$

Case 1 ($id_1 \in resources \wedge id_2 \in users$)**.**

$\Longrightarrow$ *[$id_1 \in resources \wedge id_2 \in users$ holds]*

$id_1 \in resources \wedge id_2 \in users \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V)$

$\Longrightarrow$ *[Theorem 19 and Theorem 20]*

$id_1 \in resources \wedge id_2 \in users \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$resourceProperty(id_1, V) \wedge userProperty(id_2, V)$

$\Longrightarrow$ *[Definition 13 and Definition 12]*

$id_1 \in resources \wedge id_2 \in users \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$(AcquiredResource(f_1, id_1, V) \vee ReleasedResource(f_1, id_1, V)) \wedge$
$(UserAcquiring(f_2, id_2, V) \vee UserReleasing(f_2, id_2, V))$

Case 1.1 ($AcquiredResource(f_1, id_1, V)$ holds)**.**

$\Longrightarrow$ *[$AcquiredResource(f_1, id_1, V)$ holds]*

$id_1 \in resources \wedge id_2 \in users \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$AcquiredResource(f_1, id_1, V)$

$\Longrightarrow$ *[AcquiredResource(f_1, id_1, V) def]*

$id_1 \in resources \wedge id_2 \in users \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$(odd(f.s) \wedge$
$(\exists\, id_{u1} : users(id) \bullet odd(f.s \restriction \{acquire(id_{u1}, id), release(id_{u1}, id)\}) \wedge$
$\forall\, id_{u2} : users(id) \bullet id_{u1} \neq id_{u2} \Rightarrow$
$even(f.s \restriction \{acquire(id_{u1}, id), release(id_{u1}, id)\})))$

We consider two cases for the id_{u1} in the definition of AcquiredResource(f_1, id_1, V) predicate:

- $id_{u1} = id_2$
- $id_{u1} \neq id_2$

Case 1.1.1. $id_{u1} = id_2$

$\Longrightarrow$ *[$id_{u1} = id_2$]*

$id_1 \in resources \wedge id_2 \in users \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$odd(f_1.s \restriction \{acquire(id_2, id_1), release(id_2, id_1)\}))$

[$f_1.s \restriction \{acquire(id_2, id_1), release(id_2, id_1)\} = f_2.s \restriction \{acquire(id_2, id_1), release(id_2, id_1)$]

$\Longrightarrow$

$id_1 \in resources \wedge id_2 \in users \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$\wedge\, odd(f_1.s \restriction \{acquire(id_2, id_1), release(id_2, id_1)\})$
$\wedge\, odd(f_2.s \restriction \{acquire(id_2, id_1), release(id_2, id_1)\})$

$\Longrightarrow$ *[P and r def]*

$id_1 \in resources \wedge id_2 \in users \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$g(id_1) = id_1 \wedge id_1 \in r(id_2, f_2.s)$

$\Longrightarrow$ *[$>_{RA}$ def]*

$id_1 \in resources \wedge id_2 \in users \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$g(id_1) = id_1 \wedge id_1 >_{RA} g(id_2)$

$\Longrightarrow$ *[PC]*

$g(id_1) >^*_{RA'} g(id_2)$

Case 1.1.2. $id_u \neq id_2$

$$\Longrightarrow \qquad \textit{[}id_u \neq id_2 \textit{ holds]}$$

$$id_1 \in resources \wedge id_2 \in users \wedge$$
$$state(\sigma, V) \wedge max(\sigma, V) \wedge$$
$$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$$
$$(\exists\, id_u : users \bullet id_u \neq id_2 \wedge (A(id_1, V) \setminus f_1.R) = \{release(id_u, id_1)\})$$

$$\Longrightarrow \qquad [A(id_1, V) \cap A(id_2, V) = \{acquire(id_2, id_1), release(id_2, id_1)\}]$$

$$id_1 \in resources \wedge id_2 \in users \wedge$$
$$state(\sigma, V) \wedge max(\sigma, V) \wedge$$
$$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$$
$$(A(id_1, V) \setminus f_1.R) \cap A(id_2) = \emptyset$$

$$\Longrightarrow \qquad \textit{[request def]}$$

$$id_1 \in resources \wedge id_2 \in users \wedge$$
$$state(\sigma, V) \wedge max(\sigma, V) \wedge$$
$$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$$
$$\neg request(\sigma, id_1, id_2, V)$$

$$\Longrightarrow \qquad \textit{[ungranted\_request def and PC]}$$

$$false$$

$$\Longrightarrow \qquad \textit{[PC]}$$

$$g(\sigma, id_1) >^*_{RA'} g(\sigma, id_2)$$

Case 1.2 ($ReleasedResource(f_1, id_1, V)$ holds)**.**

$$id_1 \in resources \wedge id_2 \in users \wedge$$
$$state(\sigma, V) \wedge max(\sigma, V) \wedge$$
$$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$$
$$even(f_1.s)$$

$$\Longrightarrow \qquad \textit{[g def]}$$

$$id_1 \in resources \wedge id_2 \in users \wedge$$
$$state(\sigma, V) \wedge max(\sigma, V) \wedge$$
$$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$$
$$g(\sigma, id_1) = big$$

$$\Longrightarrow \qquad [>_{RA} \textit{ def and g def]}$$

$$id_1 \in resources \wedge id_2 \in users \wedge$$
$$state(\sigma, V) \wedge max(\sigma, V) \wedge$$
$$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$$
$$g(\sigma, id_1) = big \wedge big >_{RA} g(\sigma, id_2)$$

$$\Longrightarrow \qquad \textit{[PC]}$$

$$g(\sigma, id_1) >_{RA} g(\sigma, id_2)$$

Case 2. $id_1 \in user \wedge id_2 \in resource$

$$\Longrightarrow \qquad \textit{[}id_1 \in user \wedge id_2 \in resource \textit{ holds]}$$
$$id_1 \in users \wedge id_2 \in resources \wedge$$
$$state(\sigma, V) \wedge max(\sigma, V) \wedge$$
$$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V)$$
$$\Longrightarrow \qquad \textit{[Theorem 20]}$$
$$id_1 \in users \wedge id_2 \in resources \wedge$$
$$state(\sigma, V) \wedge max(\sigma, V) \wedge$$
$$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$$
$$userProperty(id_1, V)$$
$$\Longrightarrow \qquad \textit{[Definition 13]}$$
$$id_1 \in users \wedge id_2 \in resources \wedge$$
$$state(\sigma, V) \wedge max(\sigma, V) \wedge$$
$$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$$
$$UserAcquiring(f_1, id_1, V) \vee UserReleasing(f_1, id_1, V)$$

Here we consider two cases for id_r in $UserAcquiring$ and $UserReleasing$ definitions:

- $eitheroneid_r = id_2$
- $bothid_r \neq id_2$

Case 2.1. $id_r \neq id_2$

$$\Longrightarrow \qquad \textit{[}userProperty(id_1, V) \textit{ and } id_r \neq id_2\textit{]}$$
$$id_1 \in users \wedge id_2 \in resources \wedge$$
$$state(\sigma, V) \wedge max(\sigma, V) \wedge$$
$$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$$
$$(\exists\, id_r : \operatorname{ran} resources(id) \bullet id_r \neq id_2 \wedge$$
$$(((A(id_1, V) \setminus f_1.R) = \{acquire(id_1, id_r)\}) \vee$$
$$(A(id_1, V) \setminus f_1.R) = \{release(id_1, id_r)\})))$$
$$\Longrightarrow \qquad \textit{[}A(id_1, V) \cap A(id_2, V) = \{acquire(id_1, id_2), release(id_1, id_2)\}) \textit{ and ST and PC]}$$
$$id_1 \in users \wedge id_2 \in resources \wedge$$
$$state(\sigma, V) \wedge max(\sigma, V) \wedge$$
$$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$$
$$(\exists\, id_r : \operatorname{ran} resources(id) \bullet id_r \neq id_2 \wedge$$
$$(((A(id_1, V) \setminus f_1.R) = \{acquire(id_1, id_r)\}) \vee$$
$$(A(id_1, V) \setminus f_1.R) = \{release(id_1, id_r)\})) \wedge$$
$$(A(id_1, V) \setminus f_1.R) \cap A(id_2, V) = \emptyset)$$

$\Longrightarrow$ *[PC]*

$id_1 \in users \wedge id_2 \in resources \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$(A(id_1, V) \setminus f_1.R) \cap A(id_2, V) = \emptyset \wedge$
$(\exists\, id_r : \operatorname{ran} resources(id) \bullet id_r \neq id_2 \wedge$
$(((A(id_1, V) \setminus f_1.R) = \{acquire(id_1, id_r)\}) \vee$
$(A(id_1, V) \setminus f_1.R) = \{release(id_1, id_r)\})))$

$\Longrightarrow$ *[PC]*

$id_1 \in users \wedge id_2 \in resources \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$(A(id_1, V) \setminus f_1.R) \cap A(id_2, V) = \emptyset$

$\Longrightarrow$ *[request def]*

$id_1 \in users \wedge id_2 \in resources \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$\neg request(\sigma, id_1, id_2, V)$

$\Longrightarrow$ *[ungranted_request def and PC]*

$false$

$\Longrightarrow$ *[PC]*

$g(id_1) >^{*}_{RA'} g(id_2)$

Case 2.2 ($id_r = id_2$)**.** *For this case, we consider all the cases that can occur between a process conforms to the userProperty and another conforms to the resourceProperty. These are:*

- $UserAcquiring(f_1, id_1, V)$ *and* $AcquiredResource(f_2, id_2, V)$
- $UserAcquiring(f_1, id_1, V)$ *and* $ReleaseResource(f_2, id_2, V)$
- $UserReleasing(f_1, id_1, V)$ *and* $AcquiredResource(f_2, id_2, V)$
- $UserReleasing(f_1, id_1, V)$ *and* $AcquiredResource(f_2, id_2, V)$
- $UserReleasing(f_1, id_1, V)$ *and* $ReleasedResource(f_2, id_2, V)$

Case 2.2.1 ($UserAcquiring(f_1, id_1, V) \wedge AcquiredResource(f_2, id_2, V)$ holds)**.**

$\Longrightarrow$ *[$UserAcquiring(f_1, id_1, V) \wedge AcquiredResource(f_2, id_2, V)$ holds]*

$id_1 \in users \wedge id_2 \in resources \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$min(r(f_1.s, id_1) \cup \{big\})' >_{RA} id_2 \wedge odd(f_2.s)$

$\Longrightarrow$ *[PC and g def]*

$min(r(f_1.s, id_1) \cup \{big\})' >_{RA} id_2 \wedge g(\sigma, id_2) = id_2 \wedge$
$g(\sigma, id_1) = min(r(f_1.s, id_1) \cup \{big\})'$

$\implies$ *[PC]*

$g(\sigma, id_1) >_{RA} g(\sigma, id_2)$

$\implies$ *[PC]*

$g(\sigma, id_1) >^*_{RA'} g(\sigma, id_2)$

Case 2.2.2 $(UserAcquiring(f_1, id_1, V) \wedge ReleaseResource(f_2, id_2, V))$**.**

$\implies$ *[$UserAcquiring(f_1, id_1, V) \wedge ReleaseResource(f_2, id_2, V)$ holds]*

$id_1 \in users \wedge id_2 \in resources \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$(A(id_1, V) \setminus f_1.R) = \{acquire(id_1, id_2)\} \wedge$
$(\forall\, id_u : users(id_2) \bullet acquire(id_u, id_2) \in (A(id_2, V) \setminus f_2.R))$

$\implies$ *[PC]*

$id_1 \in users \wedge id_2 \in resources \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$(A(id_1, V) \setminus f_1.R) = \{acquire(id_1, id_2)\} \wedge$
$acquire(id_1, id_2) \in (A(id_2, V) \setminus f_2.R)$

$\implies$ *[PC and ST]*

$id_1 \in users \wedge id_2 \in resources \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$(A(id_1, V) \setminus f_1.R) \cap (A(id_2, V) \setminus f_2.R) = \{acquire(id_1, id_2)\}$

$\implies$ *[PC and ST]*

$id_1 \in users \wedge id_2 \in resources \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$(A(id_1, V) \setminus f_1.R) \cap (A(id_2, V) \setminus f_2.R) \neq \emptyset$

$\implies$ *[ungrantedness def]*

$id_1 \in users \wedge id_2 \in resources \wedge$
$state(\sigma, V) \wedge max(\sigma, V) \wedge$
$id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge$
$\neg ungrantedness(\sigma, id_1, id_2, V)$

$\implies$ *[ungranted_request def and PC]*

$false$

$\implies$ *[PC]*

$g(\sigma, id_1) >^*_{RA'} g(\sigma, id_2)$

For the sub case $UserReleasing(f_1, id_1, V) \wedge AcquiredResource(f_2, id_2, V)$ we consider two cases for the id_{u1} quantified variable of the $AcquiredResource(f_2, id_2, V)$ definition:

- $id_{u1} = id_1$

- $id_{u1} \neq id_1$

Case 2.2.3 ($UserReleasing(f_1, id_1, V) \wedge AcquiredResource(f_2, id_2, V) \wedge id_{u1} = id_1$).

[$User\cancel{Re}leasing(f_1, id_1, V) \wedge AcquiredResource(f_2, id_2, V) \wedge id_{u1} = id_1$ *holds*]

$$
\begin{array}{ll}
id_1 \in users \wedge id_2 \in resources \wedge & \\
state(\sigma, V) \wedge max(\sigma, V) \wedge & \\
id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge & \\
(A(id_1, V) \setminus f_1.R) = \{release(id_1, id_2)\} \wedge & \\
(A(id_2, V) \setminus f_2.R) = \{release(id_1, id_2)\} & \\
\Longrightarrow & \text{[ST and PC]} \\
id_1 \in users \wedge id_2 \in resources \wedge & \\
state(\sigma, V) \wedge max(\sigma, V) \wedge & \\
id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge & \\
(A(id_1, V) \setminus f_1.R) \cap (A(id_2, V) \setminus f_2.R) = \{release(id_1, id_2)\} & \\
\Longrightarrow & \text{[ST and PC]} \\
id_1 \in users \wedge id_2 \in resources \wedge & \\
state(\sigma, V) \wedge max(\sigma, V) \wedge & \\
id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge & \\
(A(id_1, V) \setminus f_1.R) \cap (A(id_2, V) \setminus f_2.R) \neq \emptyset & \\
\Longrightarrow & \text{[ungrantedness def and PC]} \\
id_1 \in users \wedge id_2 \in resources \wedge & \\
state(\sigma, V) \wedge max(\sigma, V) \wedge & \\
id_1 \neq id_2 \wedge ungranted\_request(\sigma, id_1, id_2, V) \wedge & \\
\neg ungrantedness(\sigma, id_1, id_2, V) & \\
\Longrightarrow & \text{[ungranted\_request def and PC]} \\
false & \\
\Longrightarrow & \text{[PC]} \\
g(id_1) >^*_{RA'} g(id_2) &
\end{array}
$$

Case 2.2.4 ($UserReleasing(f_1, id_1, V) \wedge AcquiredResource(f_2, id_2, V) \wedge id_{u1} \neq id_1$).

[$User\cancel{Re}leasing(f_1, id_1, V) \wedge AcquiredResource(f_2, id_2, V) \wedge id_{u1} \neq id_1$ *holds*]

$$
\begin{array}{ll}
id_{u1} \neq id_1 \wedge & \\
(\exists\, id_{u1} : users(id) \bullet odd(f.s \upharpoonright \{acquire(id_{u1}, id), release(id_{u1}, id)\}) \wedge & \\
\forall\, id_{u2} : users(id) \bullet id_{u1} \neq id_{u2} \Rightarrow & \\
even(f.s \upharpoonright \{acquire(id_{u1}, id), release(id_{u1}, id)\})) \wedge & \\
odd(f_1.s \upharpoonright \{acquire(id_1, id_2), release(id_1, id_2)\}) & \\
\Longrightarrow & \text{[PC]} \\
even(f_2.s \upharpoonright \{acquire(id_1, id_2), release(id_1, id_2)\}) \wedge & \\
odd(f_1.s \upharpoonright \{acquire(id_1, id_2), release(id_1, id_2)\}) &
\end{array}
$$

$[f_2.s \upharpoonright \{acquire(id_1, id_2), release(id_1, id_2)\} = f_1.s \upharpoonright \{acquire(id_1, id_2), release(id_1, id_2)\}]$

$$
\begin{aligned}
&\Longrightarrow \\
&even(f_1.s \upharpoonright \{acquire(id_1, id_2), release(id_1, id_2)\}) \wedge \\
&odd(f_1.s \upharpoonright \{acquire(id_1, id_2), release(id_1, id_2)\}) \\
&\Longrightarrow && \text{[PC]} \\
&false \\
&\Longrightarrow && \text{[PC]} \\
&g(id_1) >^*_{RA'} g(id_2)
\end{aligned}
$$

Case 2.2.5 $(UserReleasing(f_1, id_1, V) \wedge ReleasedResource(f_2, id_2, V))$.

$$
\begin{aligned}
&\Longrightarrow && [UserReleasing(f_1, id_1, V) \wedge ReleasedResource(f_2, id_2, V)] \\
&odd(f_1.s \upharpoonright \{acquire(id_1, id_2), release(id_1, id_2)\}) \wedge \\
&even(f_2.s \upharpoonright \{acquire(id_1, id_2), release(id_1, id_2)\}) \wedge
\end{aligned}
$$

$[f_2.s \upharpoonright \{acquire(id_1, id_2), release(id_1, id_2)\} = f_1.s \upharpoonright \{acquire(id_1, id_2), release(id_1, id_2)\}]$

$$
\begin{aligned}
&\Longrightarrow \\
&odd(f_1.s \upharpoonright \{acquire(id_1, id_2), release(id_1, id_2)\}) \wedge \\
&even(f_1.s \upharpoonright \{acquire(id_1, id_2), release(id_1, id_2)\}) \\
&\Longrightarrow && \text{[PC]} \\
&false \\
&\Longrightarrow && \text{[PC]} \\
&g(id_1) >^*_{RA'} g(id_2)
\end{aligned}
$$

□

Lemma 13 (failures(F(P))).

$$
\begin{aligned}
&failures(F(P)) = \\
&\quad \{(\langle\rangle, X) | X \subseteq \Sigma \setminus \{acquire(idU, id) | idU : users(id)\}\} \\
&\quad \cup \{\langle acquire(idU, id)\rangle, X) | idU \in users(id) \wedge X \subseteq \Sigma \setminus \{release(idU, id)\} \\
&\quad \cup \{\langle acquire(idU, id), release(idU, id)\rangle \hat{}s, X) | (s, X) \in failures(P)\}
\end{aligned}
$$

Proof. Calculated with *failures* clauses. □

Lemma 14 (maxCandidatesfailures(F(P))).

$$
\begin{aligned}
&MCfailures(F^{n+1}(P)) = \\
&\quad \{(\langle\rangle, X) | X = \Sigma \setminus \{acquire(idU, id) | idU : users(id)\}\} \\
&\quad \cup \{\langle acquire(idU, id)\rangle, X) | idU \in users(id) \wedge X = \Sigma \setminus \{release(idU, id)\} \\
&\quad \cup \{\langle acquire(idU, id), release(idU, id)\rangle \hat{}s, X) | (s, X) \in MCfailures(F^n(P))\}
\end{aligned}
$$

Proof. Calculated with Lemma 13 and initials of F(P). □

Lemma 15.

$$
\begin{aligned}
&\forall f : MCfailures(ResourceSpec(id, V)) \bullet \\
&\quad ResourceAcquired(f, id, V) \vee ResourceRelease(f, id, V)
\end{aligned}
$$

Proof. The failures of a recursive are calculated as the least fixed point in the subset order with the following theorem. $failures(P) \mathrel{\widehat{=}} \bigcup_{n\in\mathbb{N}} failures(F^n(div))$ The MCfailures can be calculated using this result being then $MCfailures(P) \mathrel{\widehat{=}} \bigcup_0^{n\in\mathbb{N}} MCfailures(F^n(div))$. We prove our theorem then by induction of n.

Case 1. *Base case:* $f \in MCfailures(F^0(div))$

$$
\begin{array}{lr}
a \in MCFailures(div) & \textit{[Assumption]} \\
\Longrightarrow & [failures(div) = \emptyset] \\
a \in \emptyset & \\
\Longrightarrow & \textit{[ST and PC]} \\
false & \\
\Longrightarrow & \textit{[PC]} \\
ResourceAcquired(f, id, V) \vee ResourceRelease(f, id, V) &
\end{array}
$$

Case 2. *Inductive case:*

$$f \in MCfailures(F^n(div)) \Rightarrow ResourceAcquired(f, id, V) \vee ResourceRelease(f, id, V) \quad \text{(IH)}$$

$$\Longrightarrow$$

$$f \in MCfailures(F^{n+1}(div)) \Rightarrow ResourceAcquired(f, id, V) \vee ResourceRelease(f, id, V)$$

From Lemma 14, we know that the $f \in MCfailures(F^n(div))$ it must belong to one of the three sets described in this lemma. Lets call the sets (i),(ii) and (iii) respecting the order in which they appear in aforementioned lemma. Then we prove that for each membership case the property holds.

Case 2.1. $f \in (i)$

$$
\begin{array}{lr}
f \in (i) & [f \in (i) \ \textit{holds}] \\
\Longrightarrow & [(i) \ \textit{def}] \\
f = (\langle\rangle, X = \Sigma \setminus \{acquire(idU, id) | idU : users(id)\}) & \\
\Longrightarrow & \textit{[PC and ST]} \\
(even(f.s) \wedge (\forall\, id_u : users(id) \bullet acquire(id_u, id) \in (A(id, V) \setminus f.R) \wedge & \\
even(f.s \upharpoonright \{acquire(id_u, id), release(id_u, id)\}))) & \\
\Longrightarrow & \textit{[ReleasedResource def]} \\
ReleasedResource(f, id, V) & \\
\Longrightarrow & \textit{[PC]} \\
ReleasedResource(f, id, V) \vee AcquiredResource(f, id, V) &
\end{array}
$$

Case 2.2. $f \in (ii)$

$$
\begin{array}{lr}
f \in (ii) & [f \in (ii)] \\
\Longrightarrow & [(ii) \ \textit{def}] \\
f \in \{\langle acquire(idU, id)\rangle, X) | idU \in users(id) \wedge X = \Sigma \setminus \{release(idU, id)\} &
\end{array}
$$

$\Longrightarrow$ *[(ii) def]*
$(odd(f.s) \wedge$
$(\exists\, id_{u1} : users(id) \bullet odd(f.s \restriction \{acquire(id_{u1}, id), release(id_{u1}, id)\}) \wedge$
$\forall\, id_{u2} : users(id) \bullet id_{u1} \neq id_{u2} \Rightarrow$
$even(f.s \restriction \{acquire(id_{u1}, id), release(id_{u1}, id)\})))$

$\Longrightarrow$ *[AcquiredResource def]*
$AcquiredResource(f, id, V)$

$\Longrightarrow$ *[PC]*
$ReleasedResource(f, id, V) \vee AcquiredResource(f, id, V)$

Case 2.3. $f \in (iii)$

$f \in (iii)$ *[$f \in (iii)$ holds]*

$\Longrightarrow$ *[(iii) def]*
$f \in \{\langle acquire(idU, id), release(idU, id)\rangle\, \hat{}\, s, X)|(s, X) \in failures(P)\}$

$\Longrightarrow$ *[(s,X) = f']*
$f \in \{\langle acquire(idU, id), release(idU, id)\rangle\, \hat{}\, f'.s, f'.X)|f' \in MCfailures(F^n(P))\}$

$\Longrightarrow$ *[IH]*
$f \in \{\langle acquire(idU, id), release(idU, id)\rangle\, \hat{}\, f'.s, f'.X)|f' \in MCfailures(F^n(P))\} \wedge$
$(ReleasedResource(f', id, V) \vee AcquiredResource(f', id, V))$

Case 2.3.1. $(ReleasedResource(f', id, V)$ *holds*

$\Longrightarrow$ *[$(ReleasedResource(f', id, V)$ holds]*
$f \in \{\langle acquire(idU, id), release(idU, id)\rangle\, \hat{}\, f'.s, f'.X)|f' \in MCfailures(F^n(P))\} \wedge$
$ReleasedResource(f', id, V)$

$\Longrightarrow$ *[$(ReleasedResource(f', id, V)$ def]*
$f \in \{(\langle acquire(idU, id), release(idU, id)\rangle\, \hat{}\, f'.s, f'.R)|f' \in MCfailures(F^n(P))\} \wedge$
$(even(f'.s) \wedge (\forall\, id_u : users(id) \bullet acquire(id_u, id) \in (A(id, V) \setminus f'.R) \wedge$
$even(f'.s \restriction \{acquire(id_u, id), release(id_u, id)\})))$

$\Longrightarrow$ *[PC and SQT and ST]*
$(even(f.s) \wedge (\forall\, id_u : users(id) \bullet acquire(id_u, id) \in (A(id, V) \setminus f.R) \wedge$
$even(f.s \restriction \{acquire(id_u, id), release(id_u, id)\})))$

$\Longrightarrow$ *[$(ReleasedResource(f', id, V)$ def]*
$ReleasedResource(f, id, V)$

$\Longrightarrow$ *[PC]*
$ReleasedResource(f, id, V) \vee AcquiredResource(f, id, V)$

Case 2.3.2 $(AcquiredResource(f', id, V)$ holds)**.**

$\Longrightarrow$ *[$AcquiredResource(f', id, V)$ holds]*
$f \in \{\langle acquire(idU, id), release(idU, id)\rangle\, \hat{}\, f'.s, f'.X)|f' \in MCfailures(F^n(P))\} \wedge$
$AcquiredResource(f', id, V)$

$\Longrightarrow$ *[AcquiredResource(f′, id, V) def]*

$f \in \{\langle acquire(idU, id), release(idU, id)\rangle \hat{~} f'.s, f'.X) | f' \in MCfailures(F^n(P))\} \wedge$
$(odd(f'.s) \wedge$
$(\exists\, id_{u1} : users(id) \bullet odd(f'.s \upharpoonright \{acquire(id_{u1}, id), release(id_{u1}, id)\}) \wedge$
$\forall\, id_{u2} : users(id) \bullet id_{u1} \neq id_{u2} \Rightarrow$
$even(f'.s \upharpoonright \{acquire(id_{u1}, id), release(id_{u1}, id)\})))$

$\Longrightarrow$ *[PC and SQT and ST]*

$(odd(f.s) \wedge$
$(\exists\, id_{u1} : users(id) \bullet odd(f.s \upharpoonright \{acquire(id_{u1}, id), release(id_{u1}, id)\}) \wedge$
$\forall\, id_{u2} : users(id) \bullet id_{u1} \neq id_{u2} \Rightarrow$
$even(f.s \upharpoonright \{acquire(id_{u1}, id), release(id_{u1}, id)\})))$

$\Longrightarrow$ *[PC]*

$AcquiredResource(f, id, V)$

$\Longrightarrow$ *[PC]*

$AcquiredResource(f, id, V) \vee ReleasedResource(f, id, V)$

$\square$

Lemma 16 (failures(F(P)) where F = UserSpec)**.** *Let $AS = acquireSeq(resources(id))$ and $RS = releaseSeq(resources(id))$*

$$failures(F(P)) =$$
$$\{(s, X) | s < AS \wedge X \subseteq \Sigma \setminus \{AS(\#s)\}\}$$
$$\cup \{(AS \hat{~} s, X) | s < RS \wedge X \subseteq \Sigma \setminus \{RS(\#s)\}\}$$
$$\cup \{(AS \hat{~} RS \hat{~} s, X) | (s, X) \in failures(P)\}$$

where:

- $acquireSeq(id, s) \mathrel{\widehat=} acquire(id, head(s)) \hat{~} acquireSeq(id, tail(s))$
- $acquireSeq(id, \langle\rangle) \mathrel{\widehat=} \langle\rangle$
- $releaseSeq(id, s) \mathrel{\widehat=} release(id, head(s)) \hat{~} releaseSeq(id, tail(s))$
- $releaseSeq(id, \langle\rangle) \mathrel{\widehat=} \langle\rangle$

Proof. Calculated with the failures clauses. $\square$

Lemma 17 (MCfailures(F(P)) where F = UserSpec)**.** *Let $AS = acquireSeq(resources(id))$ and $RS = releaseSeq(resources(id))$*

$$MCfailures(F(P)) =$$
$$\{(s, X) | s < AS \wedge X = \Sigma \setminus \{AS(\#s)\}\}$$
$$\cup \{(AS \hat{~} s, X) | s < RS \wedge X = \Sigma \setminus \{RS(\#s)\}\}$$
$$\cup \{(AS \hat{~} RS \hat{~} s, X) | (s, X) \in MCfailures(P)\}$$

Proof. Calculated with MCFailures definition and Lemma 16. $\square$

Lemma 18. *Let V be an arbitrary network and id an arbitrary id of such a network.*

$$\forall f : MCfailures(UserSpec(id, V)) \bullet UserReleasing(f, id, V) \vee UserAcquiring(f, id, V)$$

Proof. The failures of a recursive are calculated as the least fixed point in the subset order with the following theorem. $failures(P) \mathrel{\widehat{=}} \bigcup_{n \in \mathbb{N}} failures(F^n(div))$ The MCfailures can be calculated using this result being then $MCfailures(P) \mathrel{\widehat{=}} \bigcup_0^{n \in \mathbb{N}} MCfailures(F^n(div))$. We prove our theorem then by induction of n.

Case 1. *Base case: $f \in MCfailures(F^0(div))$*

$$
\begin{array}{lr}
a \in MCfailures(div) & \\
\implies & [failures(div) = \emptyset] \\
a \in \emptyset & \\
\implies & [ST] \\
false & \\
\implies & [PC] \\
UserAcquiring(f, id, V) \vee UserReleasing(f, id, V) &
\end{array}
$$

Case 2. *Inductive case:*

$$f \in MCfailures(F^n(div)) \Rightarrow ResourceAcquired(f, id, V) \vee ResourceRelease(f, id, V)$$
$$\implies$$
$$f \in MCfailures(F^{n+1}(div)) \Rightarrow ResourceAcquired(f, id, V) \vee ResourceRelease(f, id, V) \tag{IH}$$

From Lemma 17, we know that the $f \in MCfailures(F^n(div))$ it must belong to one of the three sets described in this lemma. Lets call the sets (i),(ii) and (iii) respecting the order in which they appear in aforementioned lemma. Then we prove that for each membership case the property holds.

Case 2.1 $(f \in (i))$.

$$
\begin{array}{lr}
\implies & [f \in (i) \ holds] \\
f \in (i) & \\
\implies & [(i) \ def] \\
f \in \{(s, X) | s < AS \wedge X = \Sigma \setminus \{AS(\#s)\}\} & \\
\implies & [ST and SQT and PC] \\
\exists id_r : resources \bullet ((A(id, V) \setminus f.R) = \{acquire(id, id_r)\} \wedge & \\
even(f.s \upharpoonright \{acquire(id, id_r), release(id, id_r)\}) \wedge & \\
min(r(f.s, id) \cup \{big\})) >^*_{RA'} id_r & \\
\implies & [UserAcquiring \ def] \\
UserAcquiring(f, id, V) &
\end{array}
$$

$\Longrightarrow$ *[PC]*
$UserAcquiring(f, id, V) \lor UserReleasing(f, id, V)$

Case 2.2 $(f \in (ii))$**.**

$f \in (ii)$
$\Longrightarrow$ *[(ii) def]*
$f \in \{(AS\hat{\ }s, X) | s < RS \land X = \Sigma \setminus \{RS(\#s)\}\}$
$\Longrightarrow$ *[PC and SQT and ST]*
$\exists\, id_r : resources \bullet$
$((A(id, V) \setminus f.R) = \{release(id, id_r)\} \land$
$odd(f.s \restriction \{acquire(id, id_r), release(id, id_r)\}))$
$\Longrightarrow$ *[UserReleasing def]*
$UserReleasing(f, id, V)$
$\Longrightarrow$ *[PC]*
$UserReleasing(f, id, V) \lor UserAcquiring(f, id, V)$

Case 2.3 $(f \in (iii))$**.**

$\Longrightarrow$ *[$f \in (iii)$ holds]*
$f \in (iii)$
$\Longrightarrow$ *[(iii) def]*
$f \in \{(AS\hat{\ }RS\hat{\ }s, X) | (s, X) \in MCfailures(P)\}$
$\Longrightarrow$ *[f = (s,X)]*
$f \in \{(AS\hat{\ }RS\hat{\ }f'.s, f'.R) | f' \in MCfailures(P)\}$
$\Longrightarrow$ *[IH]*
$f \in \{(AS\hat{\ }RS\hat{\ }f'.s, f'.R) | f' \in MCfailures(P)\} \land$
$UserReleasing(f', id, V) \lor UserAcquiring(f', id, V)$

Case 2.3.1 $(UserReleasing(f', id, V))$**.**

$\Longrightarrow$ *[$UserReleasing(f', id, V)$ holds]*
$f \in \{(AS\hat{\ }RS\hat{\ }f'.s, f'.R) | f' \in MCfailures(P)\} \land$
$UserReleasing(f', id, V)$
$\Longrightarrow$ *[$UserReleasing(f', id, V)$ def]*
$f \in \{(AS\hat{\ }RS\hat{\ }f'.s, f'.R) | f' \in MCfailures(P)\} \land$
$\exists\, id_r : resources \bullet$
$((A(id, V) \setminus f'.R) = \{release(id, id_r)\} \land$
$odd(f'.s \restriction \{acquire(id, id_r), release(id, id_r)\}))$
$\Longrightarrow$ *[ST and SQT and PC]*
$\exists\, id_r : resources \bullet$
$((A(id, V) \setminus f.R) = \{release(id, id_r)\} \land$
$odd(f.s \restriction \{acquire(id, id_r), release(id, id_r)\}))$
$\Longrightarrow$ *[UserReleasing def]*
$UserReleasing(f, id, V)$

$\Longrightarrow$ *[PC]*
$UserReleasing(f, id, V) \lor UserAcquiring(f, id, V)$

Case 2.3.2 $(UserAcquiring(f', id, V))$**.**

$\Longrightarrow$ *[$UserAcquiring(f', id, V)$ holds]*
$f \in \{(AS \hat{} RS \hat{} f'.s, f'.R) | f' \in MCfailures(P)\} \land$
$UserAcquiring(f', id, V)$

$\Longrightarrow$ *[$UserAcquiring(f', id, V)$ def]*
$f \in \{(AS \hat{} RS \hat{} f'.s, f'.R) | f' \in MCfailures(P)\} \land$
$(\exists\, id_r : resources \bullet ((A(id, V) \setminus f'.R) = \{acquire(id, id_r)\} \land$
$even(f'.s \upharpoonright \{acquire(id, id_r), release(id, id_r)\}) \land$
$min(r(f'.s, id) \cup \{big\})) >_{RA} id_r)$

$\Longrightarrow$ *[ST and SQT and PC]*
$\exists\, id_r : resources \bullet ((A(id, V) \setminus f.R) = \{acquire(id, id_r)\} \land$
$even(f.s \upharpoonright \{acquire(id, id_r), release(id, id_r)\}) \land$
$min(r(f.s, id) \cup \{big\})) >_{RA} id_r$

$\Longrightarrow$ *[$UserAcquiring(f, id, V)$ def]*
$UserAcquiring(f, id, V)$

$\Longrightarrow$ *[PC]*
$UserAcquiring(f, id, V) \lor UserReleasing(f, id, V)$

$\square$

Theorem 19 (Resources have resourceProperty)**.** $\forall\, id : resources \bullet resourceProperty(id, V)$

Proof.

$id \in resources$ [As1]

$\Longrightarrow$ [$BehaviourRA$ restriction]
$id \in resources \land$
`ResourceSpec(id,V) [F=` $Abs(id, V)$

$\Longrightarrow$ [Theorem 10]
$id \in resources \land$
$Mfailures(Abs(id, V)) \subseteq MCfailures(ResourceSpec(id, V))$

$\Longrightarrow$ [Lemma 15]
$id \in resources \land$
$Mfailures(Abs(id, V)) \subseteq MCfailures(ResourceSpec(id, V)) \land$
$\forall\, f : MCfailures(ResourceSpec(id, V)) \bullet$
$ResourceAcquired(f, id, V) \lor ResourceRelease(f, id, V)$

$\Longrightarrow$ [PC and ST]
$id \in resources \land$
$Mfailures(Abs(id, V)) \subseteq MCfailures(ResourceSpec(id, V)) \land$
$\forall\, f : Mfailures(Abs(id, V)) \bullet$
$ResourceAcquired(f, id, V) \lor ResourceRelease(f, id, V)$

$\implies$ [PC]

$\forall f : Mfailures(Abs(id, V)) \bullet$
$ResourceAcquired(f, id, V) \lor ResourceRelease(f, id, V)$

$\implies$ [$resourceProperty(id, V)$ def]

$resourceProperty(id, V)$

$\square$

Theorem 20 (Users have userProperty)**.** $\forall id : users \bullet userProperty(id, V)$

Proof.

$id \in users$ [Assumption 1]

$\implies$ [$BehaviourRA$ restriction]

$id \in users \land$
`UserSpec(id,V) [F=` $Abs(id, V)$

$\implies$ [Theorem 10]

$id \in users \land$
$Mfailures(Abs(id, V)) \subseteq MCfailures(UserSpec(id, V))$

$\implies$ [Lemma 18]

$id \in users \land$
$Mfailures(Abs(id, V)) \subseteq MCfailures(UserSpec(id, V)) \land$
$\forall f : MCfailures(UserSpec(id, V)) \bullet UserReleasing(f, id, V) \lor UserAcquiring(f, id, V)$

$\implies$ [PC and ST]

$id \in users \land$
$Mfailures(Abs(id, V)) \subseteq MCfailures(UserSpec(id, V)) \land$
$\forall f : Mfailures(Abs(id, V)) \bullet UserReleasing(f, id, V) \lor UserAcquiring(f, id, V)$

$\implies$ [PC]

$\forall f : Mfailures(Abs(id, V)) \bullet UserReleasing(f, id, V) \lor UserAcquiring(f, id, V)$

$\implies$ [$userProperty(id, V)$ def]

$userProperty(id, V)$

$\square$

C Client/server auxiliary lemmas

Lemma 21. *Let $f(id) = \rho(\sigma, id, V)$ and let (C, σ) be a cycle of the network V such that:*

- $\forall i : \operatorname{dom} C \bullet ClientRequesting(f(C(i)), C(i), V) \lor ServerRequesting(f(C(i)), C(i), V)$

Hence, in such a cycle the following lemma holds.

$$\begin{array}{l}\forall \sigma, C \bullet Cycle(C, \sigma) \land \\ \quad (\exists i, i' : \operatorname{dom} C \bullet ClientRequesting(f(C(i)), C(i), V) \land ServerRequesting(f(C(i')), C(i'), V)) \Rightarrow \\ \qquad \exists i : \operatorname{dom} V \bullet ClientRequesting(f(C(i)), C(i), V) \land ServerRequesting(f(C(i \oplus 1)), C(i \oplus 1), V)\end{array}$$

Proof. We can prove this lemma by induction in the size of the cycle. The base case being the cycle with size 2.

Case 1 (Base case)**.** *Here, we consider the base case when the size of the cycle is zero. This is vacuously true since the predicate* $cycle(C, \sigma)$ *is false, therefore we can deduce that the desired conclusion.*

Case 2 (Inductive case)**.** *In the inductive case, we prove that if our lemma work for the case where the size of the cycle is equal to* n*, it also works to the case when the size equals to n+1. Let* (C, σ) *be a cycle where* $\#C = n$*, and a* (C', σ)*, another cycle where,* $C' = C \frown \langle id_{n+1} \rangle$ *and* $n+1$ *indicates the last position of the cycle.*

$$((\exists\, i, i' : \operatorname{dom} C \bullet \mathit{ClientRequesting}(f(C(i)), C(i), V) \wedge \mathit{ServerRequesting}(f(C(i')), C(i'), V)) \Rightarrow \qquad \text{(I.H.)}$$
$$\exists\, i : \operatorname{dom} C \bullet \mathit{ClientRequesting}(f(C(i)), C(i), V) \wedge \mathit{ServerRequesting}(f(C(i \oplus 1)), C(i \oplus 1), V))$$
$$\Longrightarrow$$
$$((\exists\, i, i' : \operatorname{dom} C' \bullet \mathit{ClientRequesting}(f(C'(i)), C'(i), V) \wedge \mathit{ServerRequesting}(f(C'(i')), C'(i'), V)) \Rightarrow$$
$$\exists\, i : \operatorname{dom} C' \bullet \mathit{ClientRequesting}(f(C'(i)), C'(i), V) \wedge \mathit{ServerRequesting}(f(C'(i \oplus 1)), C'(i \oplus 1), V))$$

Hence, we begin our reasoning by assuming the following:

[Assumption 1]
$$(\exists\, i, i' : \operatorname{dom} C' \bullet \mathit{ClientRequesting}(f(C'(i)), C'(i), V) \wedge \mathit{ServerRequesting}(f(C'(i')), C'(i'), V)$$

Here, we consider 3 cases for the for the cycle C*: the case when all the participants of the cycle behave as requesting clients, the case when all the participants behave as requesting server and when there is both a client and a server requesting in the cycle.*

- $\forall\, i : \operatorname{dom} C \bullet \mathit{ClientRequesting}(f(C'(i)), C'(i), V)$
- $\forall\, i : \operatorname{dom} C \bullet \mathit{ServerRequesting}(f(C'(i)), C'(i), V)$
- $(\exists\, i, i' : \operatorname{dom} C \bullet \mathit{ClientRequesting}(f(C(i)), C(i), V) \wedge \mathit{ServerRequesting}(f(C(i')), C(i'), V)$

Case 2.1 ($\forall\, i : \operatorname{dom} C \bullet \mathit{ClientRequesting}(f(C'(i)), C'(i), V)$)**.** *This represents the case where the* C *part of the cycle* C' *has only client requesting atoms.*

$$\Longrightarrow$$
$$\forall\, i : \operatorname{dom} C \bullet \mathit{ClientRequesting}(f(C'(i)), C'(i), V)$$
$$\Longrightarrow \qquad \text{[From Assumption 1 and Case 1.1]}$$
$$\forall\, i : \operatorname{dom} C \bullet \mathit{ClientRequesting}(f(C'(i)), C'(i), V) \wedge \mathit{ServerRequesting}(f(C'(n+1)), C'(n+1), V)$$
$$\Longrightarrow \qquad \text{[PC]}$$
$$\mathit{ClientRequesting}(f(C'(n)), C'(n), V) \wedge \mathit{ServerRequesting}(f(C'(n+1)), C'(n+1), V)$$
$$\Longrightarrow \qquad \text{[PC]}$$
$$\exists\, i : \operatorname{dom} C' \bullet \mathit{ClientRequesting}(f(C'(i)), C'(i), V) \wedge \mathit{ServerRequesting}(f(C'(i \oplus 1)), C'(i \oplus 1), V))$$

Case 2.2 ($\forall i : \operatorname{dom} C \bullet \mathit{ServerRequesting}(f(C'(i)), C'(i), V)$). *This represents the case where the C part of the cycle C' has only server requesting atoms.*

$$\Longrightarrow$$
$$\forall i : \operatorname{dom} C \bullet \mathit{ServerRequesting}(f(C'(i)), C'(i), V)$$
$$\Longrightarrow \qquad \textit{[From Assumption 1 and Case 1.1]}$$
$$\forall i : \operatorname{dom} C \bullet \mathit{ServerRequesting}(f(C'(i)), C'(i), V) \wedge \mathit{ClientRequesting}(f(C'(n+1)), C'(n+1), V)$$
$$\Longrightarrow \qquad \textit{[PC]}$$
$$\mathit{ClientRequesting}(f(C'(n+1)), C'(n+1), V) \wedge \mathit{ServerRequesting}(f(C'(1)), C'(1), V)$$
$$\Longrightarrow \qquad \textit{[PC]}$$
$$\exists i : \operatorname{dom} C' \bullet \mathit{ClientRequesting}(f(C'(i)), C'(i), V) \wedge \mathit{ServerRequesting}(f(C'(i \oplus 1)), C'(i \oplus 1), V))$$

Case 2.3 (($\exists i, i' : \operatorname{dom} C \bullet \mathit{ClientRequesting}(f(C(i)), C(i), V) \wedge \mathit{ServerRequesting}(f(C(i')), C(i'), V)$). *This represents the case where there are both a client requesting atom and a server requesting one in C.*

$$\Longrightarrow$$
$$\exists i, i' : \operatorname{dom} C \bullet \mathit{ClientRequesting}(f(C(i)), C(i), V) \wedge \mathit{ServerRequesting}(f(C(i')), C(i'), V)$$
$$\Longrightarrow \qquad \textit{[I.H.]}$$
$$\exists i : \operatorname{dom} C \bullet \mathit{ClientRequesting}(f(C(i)), C(i), V) \wedge \mathit{ServerRequesting}(f(C(i \oplus 1)), C(i \oplus 1), V))$$
$$\Longrightarrow \qquad [\operatorname{dom} C \subseteq \operatorname{dom} C']$$
$$\exists i : \operatorname{dom} C' \bullet \mathit{ClientRequesting}(f(C'(i)), C'(i), V) \wedge \mathit{ServerRequesting}(f(C'(i \oplus 1)), C'(i \oplus 1), V))$$

□

Lemma 22 (S body failures).

$$\begin{aligned}
&\mathit{failures}(F(id, V)(N)) = \\
&\quad \{(\langle\rangle, X) | req \in SRq(id) \wedge req \notin X\} \cup \\
&\quad \{(\langle req\rangle, X) | req \in SRq(id) \wedge resp \in responses(req) \wedge resp \notin X\} \cup \\
&\quad \{(\langle req\rangle \hat{\ } s, X) | req \in SRq(id) \wedge responses(req) = \emptyset \wedge (s, X) \in \mathit{failures}(S(id, V))\} \cup \\
&\quad \{(\langle req, resp\rangle, X) | req \in SRq(id) \wedge resp \in responses(req) \wedge (s, X) \in \mathit{failures}(S(id, V))\}
\end{aligned}$$

Proof. Calculated with the failures clauses. □

Lemma 23 (C body failures).

$$\begin{aligned}
&\mathit{failures}(C(id, V)) = \\
&\quad \{(\langle\rangle, X) | req \in CRq(id) \wedge req \notin X\} \cup \\
&\quad \{(\langle req\rangle, X) | req \in CRq(id) \wedge X \cap responses(req) = \emptyset\} \cup \\
&\quad \{(\langle req\rangle \hat{\ } s, X) | req \in CRq(id) \wedge responses(req) = \emptyset \wedge (s, X) \in \mathit{failures}(C(id, V))\} \cup \\
&\quad \{(\langle req, resp\rangle, X) | req \in CRq(id) \wedge resp \in responses(req) \wedge (s, X) \in \mathit{failures}(C(id, V))\}
\end{aligned}$$

Proof. Calculated with the failures clauses. $\square$

Lemma 24 (S body MCfailures)**.**

$$
\begin{aligned}
&MCfailures(S(id,V)) = \\
&\quad \{(\langle\rangle, X) | req \in SRq(id) \land req \notin X\} \cup \\
&\quad \{(\langle req\rangle, X) | req \in SRq(id) \land resp \in responses(req) \land resp \notin X\} \cup \\
&\quad \{(\langle req\rangle\hat{\ }s, X) | req \in SRq(id) \land responses(req) = \emptyset \land (s, X) \in failures(S(id,V))\} \cup \\
&\quad \{(\langle req, resp\rangle, X) | req \in SRq(id) \land resp \in responses(req) \land (s, X) \in failures(S(id,V))\}
\end{aligned}
$$

Proof. Calculated with the failures clauses plus the definition of $Mfailures$. $\square$

Lemma 25 (C body MCfailures)**.**

$$
\begin{aligned}
&failures(C(id,V)) = \\
&\quad \{(\langle\rangle, X) | req \in CRq(id) \land req \notin X\} \cup \\
&\quad \{(\langle req\rangle, X) | req \in CRq(id) \land X \cap responses(req) = \emptyset\} \cup \\
&\quad \{(\langle req\rangle\hat{\ }s, X) | req \in CRq(id) \land responses(req) = \emptyset \land (s, X) \in failures(C(id,V))\} \cup \\
&\quad \{(\langle req, resp\rangle, X) | req \in CRq(id) \land resp \in responses(req) \land (s, X) \in failures(C(id,V))\}
\end{aligned}
$$

Proof. Calculated with the failures clauses plus the definition of $Mfailures$. $\square$

Theorem 26 (S Mfailures imply pre CS property)**.**

$$
\begin{aligned}
&\forall f : failures(S(id,V)) \bullet ClientRequesting(f, id, V) \lor \\
&\quad ClientResponding(f, id, V) \lor ServerRequesting(f, id, V) \lor SReq(f, id, V)
\end{aligned}
$$

Proof.

Case 1 (Base case)**.**

$$
\begin{aligned}
&f \in failures(F^0(div)) \\
&\Longrightarrow && \text{[}F^0\ def\text{]} \\
&f \in failures(div) \\
&\Longrightarrow && \text{[}failures(div)\ def\text{]} \\
&f \in \emptyset \\
&\Longrightarrow && \text{[ST]} \\
&false \\
&\Longrightarrow && \text{[PC]} \\
&ClientRequesting(f, id, V) \lor ClientResponding(f, id, V) \lor \\
&SReq(f, id, V) \lor ServerResponding(f, id, V)
\end{aligned}
$$

Case 2 (Inductive case)**.**

$$f \in failures(F^{n+1}(div))$$

Here we split the proof since $f \in failures(F^{n+1}(div))$ *implies that* f *must belong to one of the composing set. We denote the composing sets appering in the definition of its failures by* $(i), (ii), (iii)$ *and* $(iiii)$ *respecting the order in which they appear. Hence:*

- $f \in (i)$
- $f \in (ii)$
- $f \in (iii)$
- $f \in (iiii)$

Case 2.1 ((i))**.**

$\implies$
$f \in (i)$

$\implies$ *[(i) def]*
$f \in \{(\langle\rangle, X) | req \in SRq(id) \wedge req \notin X\}$

$\implies$ *[ST and PC]*
$(f.s = \langle\rangle \vee last(f.s) \in$
$responses(id) \vee last(f.s) \in requests(id) \wedge$
$responses(last(f.s)) = \emptyset) \wedge SRq(id) \not\subseteq f.R$

$\implies$ *[SReq(f, id, V) def]*
$SReq(f, id, V)$

$\implies$ *[PC]*
$ClientRequesting(f, id, V) \vee ClientResponding(f, id, V) \vee$
$SReq(f, id, V) \vee ServerResponding(f, id, V)$

Case 2.2 ((ii))**.**

$\implies$
$f \in (ii)$

$\implies$ *[(ii) def]*
$f \in \{(\langle req\rangle, X) | req \in SRq(id) \wedge resp \in responses(req) \wedge resp \notin X\}$

$\implies$ *[ST and PC]*
$SResp(f, id, V) \wedge \exists\, ev : responses(last(f.s)) \bullet ev \in (A(id, V) \setminus f.R)$

$\implies$ *[ServerResponding(f, id, V) def]*
$ServerResponding(f, id, V)$

$\implies$ *[PC]*
$ClientRequesting(f, id, V) \vee ClientResponding(f, id, V) \vee$
$SReq(f, id, V) \vee ServerResponding(f, id, V)$

Case 2.3 ((iii)).

$$\Longrightarrow$$
$$f \in (iii)$$
$$\Longrightarrow \qquad \text{[(iii) def]}$$
$$f \in \{(\langle req \rangle \hat{\ } f'.s, f'.R) | req \in SRq(id) \wedge responses(req) = \emptyset \wedge$$
$$f' \in failures(ServerF^n(id, V)(div))\}$$
$$\Longrightarrow \qquad \text{[I.H.]}$$
$$f \in \{(\langle req \rangle \hat{\ } f'.s, f'.R) | req \in SRq(id) \wedge responses(req) = \emptyset \wedge$$
$$f' \in failures(ServerF^n(id, V)(div))\} \wedge$$
$$ClientRequesting(f', id, V) \vee ClientResponding(f', id, V) \vee$$
$$SReq(f', id, V) \vee ServerResponding(f', id, V)$$

Here, we have to split in 4 cases when each of the predicate holds for f'. In each case, when the predicate holds for f', it is quite straightforward to prove that it holds also to f, hence we only present the final conclusion.

$$\Longrightarrow \qquad \text{[PC]}$$
$$ClientRequesting(f, id, V) \vee ClientResponding(f, id, V) \vee$$
$$SReq(f, id, V) \vee ServerResponding(f, id, V)$$

Case 2.4 ((iiii)).

$$\Longrightarrow$$
$$f \in (iiii)$$
$$\Longrightarrow \qquad \text{[(iiii) def]}$$
$$f \in \{(\langle req, resp \rangle \hat{\ } s, X) | req \in SRq(id) \wedge$$
$$resp \in responses(req) \wedge (s, X) \in failures(S(id, V))\}$$
$$\Longrightarrow \qquad \text{[I.H.]}$$
$$f \in \{(\langle req, resp \rangle \hat{\ } f'.s, f'.R) | req \in SRq(id) \wedge$$
$$resp \in responses(req) \wedge f' \in failures(S(id, V))\} \wedge$$
$$ClientRequesting(f', id, V) \vee ClientResponding(f', id, V) \vee$$
$$SReq(f', id, V) \vee ServerResponding(f', id, V)$$

Here in the same way as in the previous case, we have to split in 4 cases when each of the predicate holds for f'. In each case, when the predicate holds for f', it is quite straightforward to prove that it holds also to f, hence we only present the final conclusion.

$$\Longrightarrow \qquad \text{[PC]}$$
$$ClientRequesting(f, id, V) \vee ClientResponding(f, id, V) \vee$$
$$SReq(f, id, V) \vee ServerResponding(f, id, V)$$

$\square$

Theorem 27 (C failures imply pre CS property).

$$\forall f : failures(C(id, V)) \bullet ClientRequesting(f, id, V) \vee$$
$$ClientResponding(f, id, V) \vee ServerRequesting(f, id, V) \vee SReq(f, id, V)$$

Proof. The reasoning presented here is very similar to the one present for demonstrating that the MCfailures of S.

Case 1 (Base case)**.**

$f \in failures(F^0(div))$
$\Longrightarrow$
$f \in failures(div)$
$\Longrightarrow$
$f \in \emptyset$
$\Longrightarrow$
$false$
$\Longrightarrow$
$ClientRequesting(f, id, V) \lor ClientResponding(f, id, V) \lor$
$SReq(f, id, V) \lor ServerResponding(f, id, V)$

Case 2 (Inductive case)**.**

$f \in failures(F^{n+1}(div))$

Here we split the proof since $f \in failures(F^{n+1}(div))$ *implies that f must belong to one of the composing set. We denote the composing sets appering in the definition of its failures by* $(i), (ii), (iii)$ *and* $(iiii)$ *respecting the order in which they appear. Hence:*

- $f \in (i)$
- $f \in (ii)$
- $f \in (iii)$
- $f \in (iiii)$

Case 2.1 ((i))**.**

$\Longrightarrow$
$f \in (i)$
$\Longrightarrow$
$f \in \{(\langle\rangle, X) | req \in CRq(id) \land req \notin X\}$
$\Longrightarrow$
$CReq(f, id, V) \land \exists\, req : CRq(id) \bullet req \in (A(id, V) \setminus f.R)$
$\Longrightarrow$
$ClientRequesting(f, id, V)$
$\Longrightarrow$
$ClientRequesting(f, id, V) \lor ClientResponding(f, id, V) \lor$
$SReq(f, id, V) \lor ServerResponding(f, id, V)$

Case 2.2 ((ii)).

$\Longrightarrow$
$f \in (ii)$

$\Longrightarrow$
$f \in \{(\langle req \rangle, X) | req \in CRq(id) \wedge X \cap responses(req) = \emptyset\}$

$\Longrightarrow$
$CResp(f, id, V) \wedge (A(id, V) \setminus f.R) = responses(last(f.s))$

$\Longrightarrow$
$ClientResponding(f, id, V)$

$\Longrightarrow$
$ClientRequesting(f, id, V) \vee ClientResponding(f, id, V) \vee$
$SReq(f, id, V) \vee ServerResponding(f, id, V)$

Case 2.3 ((iii)).

$\Longrightarrow$
$f \in (iii)$

$\Longrightarrow$
$f \in \{(\langle req \rangle \hat{\ } s, X) | req \in CRq(id) \wedge responses(req) = \emptyset \wedge (s, X) \in failures(F(id, V)^n(div))\}$

$\Longrightarrow$
$f \in \{(\langle req \rangle \hat{\ } s, X) | req \in CRq(id) \wedge$
$responses(req) = \emptyset \wedge (s, X) \in failures(F(id, V)^n(div))\} \wedge$
$ClientRequesting(f', id, V) \vee ClientResponding(f', id, V) \vee$
$SReq(f', id, V) \vee ServerResponding(f', id, V)$

Here, we have to split in 4 cases when each of the predicate holds for f'. In each case, when the predicate holds for f', it is quite straightforward to prove that the same predicate also holds for f, hence we only present the final conclusion.

$\Longrightarrow$
$ClientRequesting(f, id, V) \vee ClientResponding(f, id, V) \vee$
$SReq(f, id, V) \vee ServerResponding(f, id, V)$

Case 2.4 ((iiii)).

$\Longrightarrow$
$f \in (iiii)$

$\Longrightarrow$
$f \in \{(\langle req, resp \rangle, X) | req \in CRq(id) \wedge resp \in responses(req) \wedge (s, X) \in failures(F(id, V)^n(div))\}$

$\Longrightarrow$
$f \in \{(\langle req, resp \rangle, X) | req \in CRq(id) \wedge$
$resp \in responses(req) \wedge (s, X) \in failures(F(id, V)^n(div))\} \wedge$
$ClientRequesting(f', id, V) \vee ClientResponding(f', id, V) \vee$
$SReq(f', id, V) \vee ServerResponding(f', id, V)$

Here in the same way as in the previous case, we have to split in 4 cases when each of the predicate holds for f'. In each case, when the predicate holds for f', it is quite straightforward to prove that the same predicate also holds for f, hence we only present the final conclusion.

$$\Longrightarrow$$
$$ClientRequesting(f, id, V) \lor ClientResponding(f, id, V) \lor$$
$$SReq(f, id, V) \lor ServerResponding(f, id, V)$$

□

Lemma 28 (CS Mfailures)**.**

$$
\begin{aligned}
&failures(C(id, V)) = \\
&\quad \{(\langle\rangle, X) | req \in CRq(id) \land req \notin X\} \cup \\
&\quad \{(\langle req \rangle, X) | req \in CRq(id) \land X \cap responses(req) = \emptyset\} \cup \\
&\quad \{(\langle req \rangle \hat{\ } s, X) | req \in CRq(id) \land responses(req) = \emptyset \land (s, X) \in failures(C(id, V))\} \cup \\
&\quad \{(\langle req, resp \rangle, X) | req \in CRq(id) \land resp \in responses(req) \land (s, X) \in failures(C(id, V))\} \cup \\
&\quad \{(\langle\rangle, X) | req \in SRq(id) \land req \notin X\} \cup \\
&\quad \{(\langle req \rangle, X) | req \in SRq(id) \land resp \in responses(req) \land resp \notin X\} \cup \\
&\quad \{(\langle req \rangle \hat{\ } s, X) | req \in SRq(id) \land responses(req) = \emptyset \land (s, X) \in failures(S(id, V))\} \cup \\
&\quad \{(\langle req, resp \rangle, X) | req \in SRq(id) \land resp \in responses(req) \land (s, X) \in failures(S(id, V))\}
\end{aligned}
$$

Proof. Calculated with the revivals clauses. □

Lemma 29.

$$
\begin{aligned}
&\forall f : Mfailures(CS(id, V)) \bullet ClientRequesting(f, id, V) \lor \\
&\quad ClientResponding(f, id, V) \lor ServerRequesting(f, id, V) \lor SReq(f, id, V)
\end{aligned}
$$

Proof. The reasoning for this proof is very similar to the steps adopted for the two lemmas concerning processes S and C. □

Lemma 30.

$$
\begin{aligned}
&\forall f : Mfailures(RequestResponseSpec(id, V)) \bullet ClientRequesting(f, id, V) \lor \\
&\quad ClientResponding(f, id, V) \lor ServerRequesting(f, id, V) \lor SReq(f, id, V)
\end{aligned}
$$

Proof. This follows easily from lemmas 27, 26 and 29

□

Lemma 31 (Revivals of ServerReqSpec)**.**

$$
\begin{aligned}
&revivals(F(id, V)^{n+1}(P)) = \\
&\quad \{(\langle\rangle, X, a) | a \in \Sigma \setminus SRq(id) \land a \notin X\} \cup \\
&\quad \{(\langle\rangle, X, a) | a \in SRq(id) \land SRq(id) \cap X = \langle\rangle\} \cup \\
&\quad \{(\langle ev \rangle \hat{\ } s, X, a) | ev \in A(id, V) \land (s, X, a) \in revivals(F(id, V)^n(P))
\end{aligned}
$$

Lemma 32 (Revivals of ServerReqSpec)**.**

$$
\begin{aligned}
&MCrevivals(F(id,V)^{n+1}(P)) = \\
&\quad \{(\langle\rangle, X, a) | a \in \Sigma \setminus SRq(id) \wedge a \notin X \wedge (SRq(id) \cap X \neq \emptyset \Rightarrow X \supseteq SRq(id))\} \cup \\
&\quad \{(\langle\rangle, X, a) | a \in SRq(id) \wedge SRq(id) \cap X = \langle\rangle\} \cup \\
&\quad \{(\langle ev \rangle \hat{\ } s, X, a) | ev \in A(id,V) \wedge (s, X, a) \in MCrevivals(F(id,V)^n(P))
\end{aligned}
$$

Lemma 33.

$$
\begin{aligned}
&\forall\, r : MCrevivals(ServerRequestSpec(id,V)) \bullet \\
&\quad ServerRequesting(failure(r), id, V) \vee SRq(id) \subseteq failure(r).R
\end{aligned}
$$

Proof. Using the same argument as used for the other lemmas. □

Theorem 34 (CS predicate ensures clientServerProperty)**.** *Let V be a network such that $CS(V)$ holds.*

$$\forall\, id : \mathrm{dom}\, V \bullet clientServerProperty(id,V)$$

Proof. Let id be an arbitrary id of V.

$$
\begin{aligned}
&CS(V) \wedge id \in \mathrm{dom}\, V && \text{[Assumption 1]} \\
&\Longrightarrow && [BehaviourCS(V) \text{ and Lemma 30}] \\
&(\forall\, mf : Mfailures(Abs(id,V))) \bullet ClientRequesting(f,id,V) \vee \\
&ClientResponding(f,id,V) \vee ServerRequesting(f,id,V) \vee SReq(f,id,V)) \\
&\Longrightarrow && [BehaviourCS(V) \text{ and Lemma 33}] \\
&(\forall\, mf : Mfailures(Abs(id,V))) \bullet ClientRequesting(f,id,V) \vee \\
&ClientResponding(f,id,V) \vee ServerRequesting(f,id,V) \vee SReq(f,id,V)) \wedge \\
&(\forall\, mf : Mfailures(Abs(id,V)) \bullet ServerRequesting(f,id,V) \wedge SRq(id) \subseteq mf.R) \\
&\Longrightarrow && \text{[PC]} \\
&(\forall\, mf : Mfailures(Abs(id,V))) \bullet ClientRequesting(f,id,V) \vee \\
&ClientResponding(f,id,V) \vee ServerRequesting(f,id,V) \vee SReq(f,id,V)) \wedge \\
&(ServerRequesting(f,id,V) \vee SRq(id) \subseteq mf.R)
\end{aligned}
$$

Using predicate calculus we can distribute one clause into another. As $ServerRequesting$ in conjunction with any predicate other than $SReq(f,id,V)$ is false, and with $SReq(f,id,V)$ this last is absorbed by $ServerRequesting$. Also, the $SReq(f,id,V)$ in conjunction with $SReq$ is false, but with any other predicate it is absorbed by the predicate.We end up with:

$$
\begin{aligned}
&\Longrightarrow && \Longrightarrow \\
&\forall\, Mfailures(Abs(id,V)) \bullet ClientRequesting(f,id,V) \vee && clientServerProperty(id,V) \\
&ClientResponding(f,id,V) \vee ServerRequesting(f,id,V) \vee \\
&ServerRequesting(f,id,V)
\end{aligned}
$$

□

D CSPM models

D.1 Network definitions

```
-- Network Common

-----------------
-- Auxiliary definition for the network model
-----------------

-- Functions to recover the ID, Behavior and Alphabet given a atomic tuple.
ID_((x,y,z)) = x
B_((x,y,z)) = y
A_((x,y,z)) = z

-- Functions to recover the Alphabet and Behaviour of an atom
-- given an Id and a Network containing this id
A(id,V) = A_(getElement(id,V))
B(id,V) = B_(getElement(id,V))


-- Auxiliary functional definitions
pick({x}) = x
getElement(id,V) = pick({ a | a <- V, ID_(a) == id})

-- Function to recover the vocabulary of the network V
--Voc(V) = Union({ inter(A_(a1),A_(a2)) | a1 <- V, a2 <- V, NEQ(a1,a2)})

Voc(V) =
  let
  inters(a,<b>^ts) = union(inter(A_(a),A_(b)),inters(a,ts))
  inters(a,<>) = {}
  VocP(<a>^ts) = union(inters(a,ts),VocP(ts))
  VocP(<>) = {}
  within
  VocP(seq(V))

-- Intersection between and alphabet and the vocabulary of the network
AVoc(id,V) = let Aid = A(id,V)
    within Union({inter(Aid,A_(a)) | a <- V, ID_(a) != id})

-- Abstraction function
Abs(id,V) = B(id,V) \ diff(A(id,V),AVoc(id,V))

-- Create a network based on an Ids set and a
```

```
-- function for given the behaviour and alpha of
-- tuples
DefaultNetwork(Ids,Beh,Alp) = {(id,Beh(id),Alp(id)) | id <- Ids}

-- Function to recover the alphabet of the network V
AlphaNetwork(V) = Union({ A_(a)| a <- V})

-- Function to recover the union of every alphabetical triple joint
-- Alphabetical triple joint is given by Inter({A_(a1),A_(a2),A_(a3)}) where
-- a1,a2 and a3 are three different triples
UnionTripleJoints(V) =
    Union({ Inter({A_(a1),A_(a2),A_(a3)}) |
     a1 <- V, a2 <- V, a3 <- V,NEQ(a1,a2),NEQ(a3,a2),NEQ(a1,a3)})

-- Function that gives the behaviour of a network V
Behaviour(V) = || a : V @ [A_(a)] B_(a)

-- Auxiliary definition of not equal tuples
NEQ(A1,A2) = ID_(A1) != ID_(A2)
```

D.2 Ring buffer model

```
include "../../Network.csp"

Value = {0..2}

NCELLS = N-1
nametype CELL_IDS = {0..N-2}

datatype IDS = CELL.CELL_IDS | CONTROLLER

channel input, output: Value

---------------------------------------------------------------------
-- The controller
---------------------------------------------------------------------

Controller =
 let ControllerState(cache,size,top,bot) =
       InputController(cache,size,top,bot) [] OutputController(cache,size,top,bot)

     InputController(cache,size,top,bot) =
       size < N & input?x ->
           (size == 0 & ControllerState(x,1,top,bot)
           []
```

```
          size > 0 & write.top!x -> ControllerState(cache,size+1,(top+1)%NCELLS,bot))

     OutputController(cache,size,top,bot) =
       size > 0 & output!cache ->
          (size > 1 & (read.bot?x ->ControllerState(x,size-1,top,(bot+1)%NCELLS))
          []
          size == 1 & ControllerState(cache,0,top,bot))

 within
ControllerState(0,0,0,0)

--------------------------------------------------------------------
-- The ring
--------------------------------------------------------------------
-- A generic cell
channel read, write:  CELL_IDS. Value

RingCell(id) =
  let Cell(val) =
    read.id!val -> Cell(val) [] write.id?x -> Cell(x)
  within
    Cell(0)

-- The distributed ring
Ring = ||| i: CELL_IDS @ RingCell(i)

--------------------------------------------------------------------
-- The Buffer Network
--------------------------------------------------------------------
Be(CONTROLLER) = Controller
Be(CELL.id) = RingCell(id)
Al(CONTROLLER) = {|read,write,input,output|}
Al(CELL.id) = {|read.id,write.id|}

RingBufferNetwork = {(id,Be(id),Al(id)) | id <- IDS}
```

D.3 Dinning philosophers model

```
-- Dinning Philosophers
include "../../Network.csp"

nametype NS = {0..N-1}
datatype IDS = PHIL.NS | FORK.NS
channel sit, getup, eat : NS
```

```
channel pickup,putdown : NS.NS

next(id) = (id + 1) % N
prev(id) = (id - 1) % N

Phil(id) = sit.id -> pickup.id.id -> pickup.id!next(id) ->
     eat.id -> putdown.id.id -> putdown.id!next(id) -> getup.id -> Phil(id)
APhil(id) = sit.id -> pickup.id!next(id) -> pickup.id.id ->
     eat.id -> putdown.id!next(id) -> putdown.id.id -> getup.id -> APhil(id)

Fork(id) = [] i : {id,prev(id)} @ pickup.i.id -> putdown.i.id -> Fork(id)

Al(FORK.id) = {|pickup.i.id,putdown.i.id | i <- {id,prev(id)}|}
Al(PHIL.id) = {|pickup.id.i,putdown.id.i,sit.id,getup.id,eat.id | i <- {id,next(id)}|}

Be(FORK.id) = Fork(id)
Be(PHIL.id) = Phil(id)

DinningPhilosophersNetwork = union({(id,Be(id),Al(id)) |
     id <- diff(IDS,{PHIL.(N-1)})},{(PHIL.(N-1),APhil(N-1),Al(PHIL.(N-1)))})
```

D.4 Leader election simplified model

```
-- Leader Election Model

NODE_IDS = {0..N-1}

channel transmit : NODE_IDS.NODE_IDS.CLAIM.PRIORITY
channel requestData : NODE_IDS

Transmit(id) = requestData.id -> updateXData.id?data -> Send(id,data);Transmit(id)

Send(id,data) =
    let
        S(s) = if s != <> then transmit.id!head(s)!data -> S(tail(s))
               else SKIP
    within
        S(<0..(id-1)>^<(id+1)..N-1>)

print <0..(-1)>^<1..1>

Receive(id) =
    transmit?idR!id?data ->  updateRData.id!idR!data -> Receive(id)

Control(id,data) =
```

```
    (requestData.id -> updateXData.id!data -> Control(id,data)
    []
    updateRData.id?idR?d -> updateMemData.id!idR!d -> Election(id,data))
    |~|
    Init(id)

Init(id) = reset.id -> Control(id,undecided.0)

Election(id,claim.priority) =
    readLeaders!id -> getLeaders.id?leaders ->
    readHighestPriority!id -> getHighestPriority.id?hPri ->
    readHighestPriorityId!id -> getHighestPriorityId.id?hPriId ->
    readToVote!id -> getToVote.id?toVote ->
    (if claim == leader then
        if leaders > 0 then
            Init(id)
        else
            Control(id,claim.priority)
    else if claim == follower then
        if leader == 0 then
            Init(id)
        else
            Control(id,claim.priority)
    else
        if leaders > 0 then
            Control(id,follower.priority)
        else if toVote == 0 then
            if hPri < priority or (hPri == priority and hPriId < id) then
                Control(id,leader.priority)
            else
                Control(id,follower.priority)
        else
            Control(id,claim.priority))

-- Hp : highest priority
-- hPId: highest priority id

datatype CLAIM = leader | follower | undecided
nametype PRIORITY = { -1,0,1}
channel reset : NODE_IDS
channel getLeaders : NODE_IDS.{0..N}
channel getHighestPriority : NODE_IDS.PRIORITY
channel getHighestPriorityId : NODE_IDS.NODE_IDS
channel getToVote : NODE_IDS.{0..N}
channel readToVote,readHighestPriority,readLeaders,readHighestPriorityId : NODE_IDS
channel updateMemData,updateRData : NODE_IDS.NODE_IDS.CLAIM.PRIORITY
```

```
channel updateXData : NODE_IDS.CLAIM.PRIORITY

Memory(id) =
    let
        Mem =
            readLeaders!id -> (|~| leaders : {0..N} @ getLeaders!id!leaders -> Mem)
            []
            readHighestPriority!id -> (|~| hP : {0,1} @ getHighestPriority!id!hP -> Mem)
            []
            readHighestPriorityId!id ->
                (|~| hPId : NODE_IDS @ getHighestPriorityId!id!hPId -> Mem)
            []
            readToVote!id -> (|~| toVote : {0..N} @ getToVote!id!toVote -> Mem)
            []
            updateMemData!id?idR?newC?newP -> Mem
        within
            Mem


-----------------------------------
-- Leader Election Network
-----------------------------------
include "../../Network.csp"

datatype IDS = TRANSMIT.NODE_IDS | RECEIVE.NODE_IDS | CONTROL.NODE_IDS | MEMORY.NODE_IDS

Al(TRANSMIT.id) = {|updateXData.id,transmit.id,requestData.id|}
Al(RECEIVE.id) = {|transmit.idR.id, updateRData.id | idR <- NODE_IDS|}
Al(CONTROL.id) = {|updateRData.id, requestData.id,updateXData.id,
                    updateMemData.id, reset.id, getLeaders.id,
                    getHighestPriority.id,getHighestPriorityId.id,
                    getToVote.id, readToVote.id,readHighestPriority.id,
                    readLeaders.id,readHighestPriorityId.id|}
Al(MEMORY.id) = {|updateMemData.id, getLeaders.id, getHighestPriority.id,
                   getHighestPriorityId.id, getToVote.id,
                   readToVote.id,readHighestPriority.id,readLeaders.id,
                   readHighestPriorityId.id|}

Be(TRANSMIT.id) = Transmit(id)
Be(RECEIVE.id) = Receive(id)
Be(CONTROL.id) = Init(id)
Be(MEMORY.id) = Memory(id)

print Al(MEM_CELL.0.1)

Network = DefaultNetwork(IDS,Be,Al)
```

Acknowledgments

The EU Framework 7 Integrated Project COMPASS (Grant Agreement 287829) financed most of the work presented here.

References

[1] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, 1993.

[2] Saddek Bensalem, Andreas Griesmayer, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. D-finder 2: Towards efficient correctness of incremental design. In *NASA Formal Methods*, pages 453–458, 2011.

[3] Stephen D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 281–305. Springer, 1984.

[4] Stephen D. Brookes and A. W. Roscoe. Deadlock analysis in networks of communicating processes. *Distributed Computing*, 4:209–230, 1991.

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

[6] J. M. R. Martin and P. H. Welch. A Design Strategy for Deadlock-Free Concurrent Systems. *Transputer Communications*, 3(4):215–232, 1997.

[7] Jeremy Martin. *Deadlock checker repository*, 2012. `http://wotug.org/parallel/theory/formal/csp/Deadlock/`.

[8] Rodrigo Ramos, Augusto Sampaio, and Alexandre Mota. Systematic development of trustworthy component systems. In *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2009.

[9] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.

[10] A. W. Roscoe and Naiem Dathi. The pursuit of deadlock freedom. *Inf. Comput.*, 75(3):289–327, 1987.

[11] A. W. Roscoe, Paul H. B. Gardiner, Michael Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking csp or how to check 10^{20} dining philosophers for deadlock. In *TACAS*, volume 1019 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 1995.

[12] A.W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[13] University of Oxford. *FDR: User Manual, version 2.94*, 2012. http://www.cs.ox.ac.uk/projects/concurrency-tools/.