A Memory Module Specification using Composable High-level Petri Nets

Eike Best

Institut für Informatik, Universität Hildesheim,
Marienburger Platz 22, D-31141 Hildesheim,
e.best@informatik.uni-hildesheim.de

Abstract. This paper describes a solution of the RPC-memory specification problem defined by M. Broy and L. Lamport on the occasion of a Dagstuhl seminar in 1994. The approach is based on a recently developed model of composable high-level Petri nets called M-nets, on which CCS-like composition operations such as parallel composition, restriction and transition synchronisation are defined. Correctness proofs are given in terms of interface transition trace inclusion.

1 Introduction

The high-level Petri net model devised by H.J. Genrich, K. Jensen [8, 11] and others can be viewed as a means of abstracting and abbreviating large elementary nets. A standard operation defined on a high-level net is its unfolding into an elementary net. In the M-net model proposed in [4], a class of high-level nets is enriched with CCS-like composition operations [16] such as choice, parallel composition, transition synchronisation and restriction. Synchronisation and restriction are driven by parametrised actions generalising the action algebra of CCS. It is shown in [4, 12] that all operations are coherent with respect to unfolding, in the sense that unfolding a composed net amounts to composing the individual unfoldings.

This paper presents an M-net solution of the RPC-memory specification problem defined in [6]. The problem asks for the modelling of a set of components to access remote memory. A remote user can request a service by issuing an RPC (remote procedure call) with parameters *read* or *write* and further parameters specifying a location and (in case of a *write*) a value. This interface calls other components performing the corresponding local *read* or *write* actions. Parametrised M-net actions can be used directly to model such procedure interfaces.

The paper is structured as follows. Section 2 introduces M-nets and a small set of operations needed for solving the RPC-memory specification problem. In section 3 a relation of implementation between two M-nets is defined. Section 4 describes a multi-level M-net specification of the memory system. Section 5 presents its implementation in terms of an RPC-module and states its correctness in terms of the notion of implementation defined in section 3. In section 6 it is shown how the RPC-module can be implemented by a timed lossy module. Section 7 contains some discussion. Proofs can be found in the appendix.

2 The M-net model

Section 2.1 defines structure and markings of an M-net. Section 2.2 defines transition rule and unfolding of M-nets. Section 2.3 describes some composition operations on M-nets.

2.1 Definition of M-nets

Let $a, b, c, \ldots$ be variables and let $v, w, \ldots$ be values. An *M-net* is a structure $N = (S, T, \iota)$ where, as usual [20], S is the set of *places* and T is the set of *transitions* (satisfying $S \cap T = \emptyset$). In the following, we use letters $s, \ldots$ to denote places and $t, \ldots$ to denote transitions. The *inscription* function ι associates:

- With every place $s \in S$, a *label* $\lambda(s)$ (as described below) and a nonempty set of values (also called its *color* [11] or *type*) $\alpha(s)$. We use the pair notation $\iota(s) = (\lambda(s)|\alpha(s))$ to separate label from type.
- With every pair (s, t) and (t, s) (for $s \in S$ and $t \in T$), a multiset of variables or values.
- With every transition t, a pair $\iota(t) = (\lambda(t)|\alpha(t))$ where $\lambda(t)$ is its *label* (as described below) and $\alpha(t)$ is a predicate (also called the *guard* of t).

For a transition t, the set $Var(t)$ is defined as the union of all variables occurring in multisets $\iota(s, t)$ and $\iota(t, s')$ (for all $s, s' \in S$) or occurring freely in $\iota(t)$. We show later that, without loss of generality, it may be assumed that ι satisfies

$$(s \neq s' \Rightarrow \alpha(s) \cap \alpha(s') = \emptyset) \quad \text{and} \quad (t \neq t' \Rightarrow Var(t) \cap Var(t') = \emptyset). \qquad (1)$$

However, for succinctness, we will not enforce this property in our examples.

The label $\lambda(s)$ is, by definition, an element of $\{\mathbf{e}, \mathbf{i}, \mathbf{x}\}$. If $\lambda(s) = \mathbf{e}$ then s is *initial*, if $\lambda(s) = \mathbf{i}$ then s is *internal*, and if $\lambda(s) = \mathbf{x}$ then s is *final*. $\lambda(s)$ drives place-based composition operations. For instance, composing two nets sequentially means that the final places of the first are combined with the initial places of the second and become internal places in the combined net [3, 17]. A typical place inscription is $\iota(s) = (\mathbf{i} \mid \{0, 1, 2\})$. This means that s is internal and may hold tokens 0, 1 or 2.

The label $\lambda(t)$ is, by definition, a multiset of *parametrised actions*. If $\lambda(t) \neq \emptyset$ then t is called *external*, or an *interface transition*, otherwise it is called *internal*. We distinguish interface transitions by representing them pictorially by the double-square symbol ⧈, reserving the single-square symbol □ for internal transitions. A parametrised action is a term of the form $A(a_1, \ldots, a_m)$ where A is an *action* *symbol* and $a_1, \ldots, a_m$ are either values or variables. We assume that there is a sufficiently large supply of action symbols and that every one of them has an arity ar (such as m for A above). Furthermore we assume that there exists a bijection $\bar{}$ called *conjugation* from action symbols to action symbols satisfying

$$\forall A\colon\ (\overline{A} \neq A) \wedge (\overline{\overline{A}} = A) \wedge (ar(\overline{A}) = ar(A)).$$

For example, if $m=2$ then $A(a,a)$, $\overline{A}(b,2)$ and $\overline{A}(2,3)$ are parametrised actions. A parametrised action is *constant* (or a *ground action*) if all of its parameters are values. The action algebra of CCS [16] is a special case of the above with all arities being zero. In the sequel we use upper-case letters, or words containing upper-case letters, to denote action symbols. $\lambda(t)$ drives transition-based composition operations. For instance, as in CCS, transition synchronisation is driven by conjugate action symbols in the labels of two transitions. A typical transition inscription is $\iota(t) = (\,\{A(a,b), \overline{B}(b,a)\} \mid a{=}5 \wedge b{\geq}0\,)$. This indicates that for the enabling of t, a binding of variables in $Var(t)$ has to be found. Under such a binding, the label and the guard of t evaluate to a multiset of ground actions and a truth value, respectively. For example, under the binding $a \mapsto 1, b \mapsto 2$, the above inscription evaluates to $(\,\{A(1,2), \overline{B}(2,1)\} \mid \mathbf{false}\,)$.

A *marking* M of N is a function associating with every place s a finite multiset $M(s)$ over $\alpha(s)$. When an initially marked M-net $\Sigma = (S, T, \iota, M^0)$ is given we refer to the set of its transitions by $T_\Sigma{=}T$ and to its initial marking by $M^0_\Sigma{=}M^0$. Figure 1 shows three marked M-nets. For instance, the net Σ_3 on the right-hand side is $(\{s_1, s_2\}, \{t_3\}, \iota, M^0_3)$ with

$$\begin{array}{llll}
\lambda(s_1) & = \mathbf{e} & \alpha(s_1) & = \{\bullet\} \\
\lambda(s_2) & = \mathbf{x} & \alpha(s_2) & = \{\bullet\} \\
\lambda(t_3) & = \{X(c,d)\} & \alpha(t_3) & = (d{=}1{-}c) \\
\iota(s_1,t_3) & = \{a\} & \iota(t_3,s_1) & = \emptyset \\
\iota(s_2,t_3) & = \emptyset & \iota(t_3,s_2) & = \{b\} \\
M^0_3(s_1) & = \{\bullet\} & M^0_3(s_2) & = \emptyset.
\end{array}$$

We also have $Var(t_1){=}\{a,b\}$, $Var(t_2){=}\{a,b,c,d,e,f\}$ and $Var(t_3){=}\{a,b,c,d\}$.

2.2 Transition rule and unfolding

A *binding of* t, where t is a transition, is a function σ associating a value with every variable in $Var(t)$.[1] A binding of t is *legal* if variables get assigned values from the types of the places they correspond to, i.e. if b is a variable occurring in $\iota(s,t)$ then $\sigma(b){\in}\iota(s)$ and if a is a variable occurring in $\iota(t,s')$ then $\sigma(a){\in}\iota(s')$. In what follows we will tacitly assume all bindings to be legal, unless specified otherwise. A binding σ of t is *enabling (for t)* if the predicate $\iota(t)$ evaluates to **true** under σ. A marking M *activates t under σ (or 'in mode σ')* if σ enables

[1] In the following, we use standard notation relating to bindings: if $\mathcal{O}$ is some object then $\mathcal{O}[\sigma]$ denotes $\mathcal{O}$ where σ is applied (elementwise) to $\mathcal{O}$.

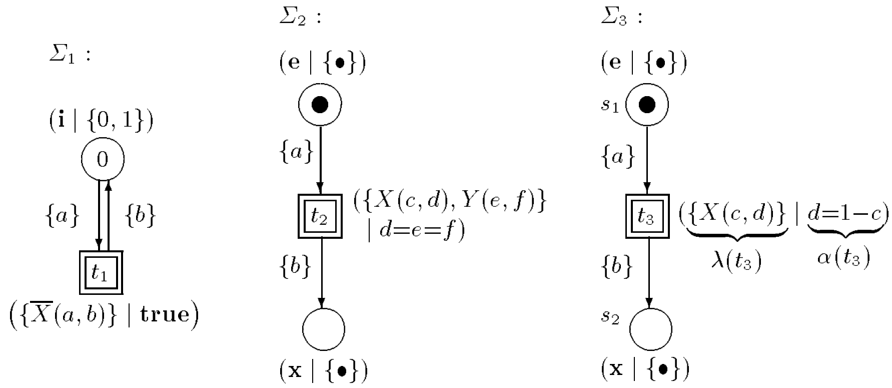


Figure 1. Three marked M-nets Σ_1 (left), Σ_2 (middle) and Σ_3 (right)

t and, moreover, for all places s, the multiset of values $\iota(s,t)[\sigma]$ (i.e. σ applied to every individual element of $\iota(s,t)$) is a sub-multiset of $M(s)$. If M activates t in mode σ then t may *occur in mode* σ yielding a successor marking M' which is calculated as follows (using multiset difference $-$ and multiset sum $+$):

$$M'(s) \quad = \quad (M(s) - (\iota(s,t)[\sigma])) + (\iota(t,s)[\sigma])$$

for every place s. In this formula, $\iota(s,t)[\sigma]$ and $\iota(t,s)[\sigma]$ denote the tokens *consumed* and *produced*, respectively, on place s. We also write $M \xrightarrow{t:\sigma} M'$ if M activates t in mode σ and M and M' are related as above.

To illustrate these definitions, consider figure 1 and three bindings for transition t_3 of Σ_3 (recall that $Var(t_3){=}\{a,b,c,d\}$):

$$\begin{aligned}
\sigma_1 &: \ a \mapsto \bullet, b \mapsto 0, c \mapsto 1, d \mapsto 0 \\
\sigma_2 &: \ a \mapsto \bullet, b \mapsto \bullet, c \mapsto 1, d \mapsto 1 \\
\sigma_3 &: \ a \mapsto \bullet, b \mapsto \bullet, c \mapsto 1, d \mapsto 0.
\end{aligned}$$

The first binding σ_1 is not legal because the value of b is not in the type of the corresponding place s_2. The second binding is legal but not enabling for t_3 because predicate $\alpha(t_3)$ does not evaluate to **true**. The third binding σ_3 is legal and enabling for t_3, and, moreover, the initial marking M_3^0 of Σ_3 activates t_3 in mode σ_3. After the occurrence of t_3 in mode σ_3, M_3^0 is changed into the marking M_3^1 defined by $M_3^1(s_1){=}\emptyset$ and $M_3^1(s_2){=}\{\bullet\}$. Note that σ_3 is still legal and enabling for t_3, but M_3^1 does not activate t_3 in mode σ_3 (due to there being insufficiently many tokens $\bullet$ on place s_1).

As another example, in the net Σ_2 shown in the middle of figure 1,

$$a \mapsto \bullet, b \mapsto \bullet, c \mapsto 0, d \mapsto 1, e \mapsto 1, f \mapsto 1$$

is a binding activating t_2 at marking $M^0_{\Sigma_2}$. If t_2 occurs under this binding, then the effect is to remove a token $\bullet$ from its input place and to add a token $\bullet$ to its output place. By contrast, the binding

$$a \mapsto \bullet, b \mapsto \bullet, c \mapsto 0, d \mapsto 1, e \mapsto 1, f \mapsto 0$$

does not enable t_2 because it makes the guard $d{=}e{=}f$ **false**. In Σ_1, the binding $a \mapsto 1, b \mapsto 1$ is legal and enabling for t_1 but does not activate t_1 under the shown marking, because there is no 1 token on the input place of t_1.

Note that whether or not a binding activates a transition t depends only on $Var(t)$. This implies that any consistent renaming of variables within $Var(t)$ (and in the inscriptions of t and its surrounding arcs) does not affect the semantics of the M-net. In particular, we may – as was said above – assume without loss of generality that $t{\neq}t'$ implies $Var(t){\cap}Var(t'){=}\emptyset$.

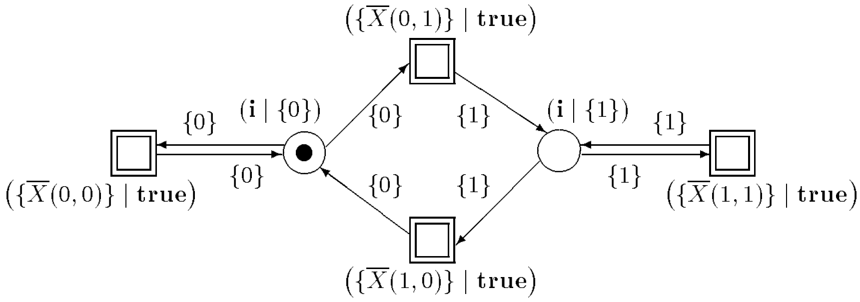


Figure 2. Unfolding $U(\Sigma_1)$ of net Σ_1 on the left-hand side of figure 1

Note also that the transition rule depends only on the types $\alpha(s)$ of places and the guards $\alpha(t)$ of transitions. In particular, it is independent of their labels λ. The same is true for unfolding, which turns an arbitrary M-net into an elementary net, where an M-net is called *elementary* (E-net for short) iff:

- Place types are singletons.
- Arc inscriptions are finite multisets over the types of the corresponding place.
- All transitions t satisfy $Var(t){=}\emptyset$ and $\alpha(t){=}\mathbf{true}$.

E-nets and arc-weighted P/T-nets [20] carrying place labels from the set $\{\mathbf{e}, \mathbf{i}, \mathbf{x}\}$ and transition labels from the set of multisets over ground actions are in 1–1 correspondence with each other. To transform a labelled arc-weighted P/T-net into an elementary M-net, attach to every place an arbitrary singleton value set (e.g. $\iota(s){=}\{s\}$ where s is its name) and to every transition a guard **true**. For arcs (s,t), add a multiset containing the only element of $\iota(s)$ as many times as specified by its weight, and do the same for arcs (t, s'). The labels remain the same. To obtain a labelled arc-weighted P/T-net from an elementary M-net, apply the converse construction, simply forgetting all value sets and replacing them by the singleton set $\{\bullet\}$ and replacing all arc inscriptions by their cardinalities.

Let $\Sigma = (S, T, \iota, M)$ be a marked M-net. Its *unfolding* $U(\Sigma) = (S_U, T_U, \iota_U, M_u)$ is defined as follows:

- $S_U = \{s_v \mid s{\in}S \text{ and } v{\in}\alpha(s)\}$,
  where for each $s_v \in S_U$: $\lambda(s_v){=}\lambda(s)$ and $\alpha(s_v){=}\{v\}$.
- $T_U = \{t_\sigma \mid t{\in}T \text{ and } \sigma \text{ is an enabling binding of } t\}$,
  where for each $t_\sigma \in T_U$: $\lambda(t_\sigma){=}(\lambda(t))[\sigma]$ and $\alpha(t_\sigma){=}(\alpha(t))[\sigma]$.
- $\iota_U(s_v, t_\sigma) = (\iota(s,t)[\sigma])\lceil_v$ and $\iota_U(t_\sigma, s_v) = (\iota(t,s)[\sigma])\lceil_v$.[2]
- For every place $s_v \in S_U$, $M_U(s_v) = M(s)\lceil_v$.

Clearly, $U(\Sigma)$ is an elementary M-net. Moreover, Σ and $U(\Sigma)$ are behaviourally equivalent in the sense that $M \xrightarrow{t:\sigma} M'$ in Σ iff $M_U \xrightarrow{t_\sigma:id} M'_U$ in $U(\Sigma)$.

For example, figure 2 shows the E-net $U(\Sigma_1)$ associated with the M-net Σ_1 on the left-hand side of figure 1. The two places of the unfolded net correspond to the two possible tokens on the place of Σ_1, namely 0 on the left and 1 on the right. The four transitions correspond to the four enabling bindings of transition t_1 in figure 1. The place corresponding to value 0 has a (single) initial token because the place in Σ_1 carries a (single) token 0.

On elementary M-nets, there exists a canonical notion of isomorphism: two (marked) E-nets are called *isomorphic* iff there exists an arc-preserving, label-preserving and marking-preserving bijection between their places and transitions. If Σ is elementary then Σ and $U(\Sigma)$ are isomorphic. Two M-nets Σ and Σ' are called *unfolding-equivalent* (in symbols: $\Sigma \equiv \Sigma'$) iff their unfoldings are isomorphic. Unfolding-equivalence can be characterised graph theoretically [9, 13] and linear algebraically [5]. To all intents and purposes, unfolding-equivalent M-nets Σ and Σ' describe the same system. For instance, consider figure 3 where Σ_1 and Σ_2 are unfolding-equivalent. Σ_3 shows their common unfolding (which is thus also unfolding-equivalent with both Σ_1 and Σ_2) and Σ_4 shows the equivalent labelled place/transition-net.

The operation of renaming variables consistently within $Var(t)$ is unfolding-preserving, and so is the operation of exchanging a value set $\alpha(s)$ with an isomorphic set $\alpha(s)'$ (modifying types and operations as necessary). Hence every M-net can be transformed into an unfolding-equivalent one satisfying (1).

2.3 Composition operations

In this section we describe three composition operations on M-nets: (disjoint) parallel composition, restriction and transition synchronisation. Two nets are combined by *disjoint parallel composition* by forming their disjoint union.[3] This operation (obviously) commutes with unfolding in the following sense:

$$U(\Sigma_1 \| \Sigma_2) \;\equiv\; U(\Sigma_1) \| U(\Sigma_2).$$

We define *restriction* to be a mixed operation $\Sigma \,\mathbf{rs}\, A$ accepting an M-net Σ and an action symbol A and returning a net in which all transitions (together

[2] Where, by definition, $\mu\lceil_v$ is the multiset containing v as many times as μ and all other elements zero times.

[3] Suitably renaming places and transitions if necessary.

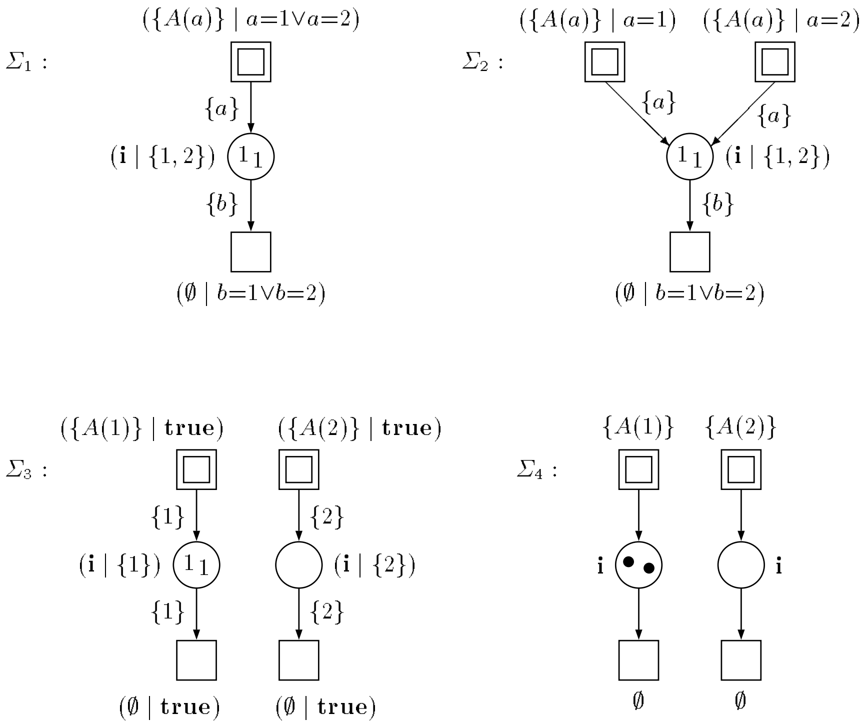


Figure 3. Illustration of the notion of unfolding-equivalence

with their surrounding arcs) whose labels contain at least one action term $A(a_1, \ldots, a_m)$ or $\overline{A}(a_1, \ldots, a_m)$ are deleted. We also need *transition synchronisation* which adds, rather than deletes, transitions to a net Σ. Following [3], synchronisation is defined as a sort of 'converse' of restriction. By definition, Σ **sy** A is the smallest net satisfying the following:

- The sets of places of Σ and Σ **sy** A are the same.
- Every transition of Σ is also a transition of Σ **sy** A.
- If t, t' are transitions of Σ **sy** A and t'' arises from t and t' by a basic synchronisation step entailed by $(A, \overline{A})$, then t'' is a transition of Σ **sy** A.

The heart of this definition is that of a basic synchronisation step [4]. We explain this definition first and illustrate it on an example later. Define t'' to arise from t and t' by a *basic synchronisation step* (entailed by $(A, \overline{A})$) if (without loss of generality) $Var(t) \cap Var(t') = \emptyset$ and, moreover,

(a) The label of t contains a term $A(\ldots)$ and the label of t' contains a conjugate term $\overline{A}(\ldots)$.
(b) Variables and constants in the parameter lists of $A(\ldots)$ and $\overline{A}(\ldots)$ are unified and renamed consistently throughout $Var(t)$ and $Var(t')$.
(c) The label of t'' equals the multiset sum of the labels of t and t', minus the two terms mentioned in (a), modulo the renamings (b).
(d) The arcs surrounding t'' are the multiset sum of the arcs surrounding t and t', modulo renamings (b).
(e) The guard of t'' is the logical conjunction of the guards of t and t', modulo renamings (b).

Intuitively speaking, this operation may be characterised as CCS-like synchronisation extended to action terms of arbitrary arity (viz. (a)). It involves *unification* (of the parameter lists of the two participating actions, viz. (b)), *multiset sum* (of labels, (c), and arcs, (d)) and *conjunction* (of guards, (e)).

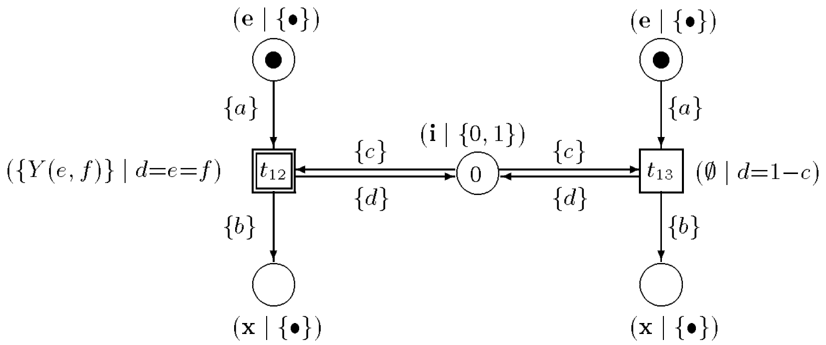


Figure 4. Synchronisation and restriction of the nets in figure 1 over X

As an example, figure 4 shows the net $((\Sigma_1\|\Sigma_2\|\Sigma_3)\ \mathbf{sy}\, X)\ \mathbf{rs}\, X$ where Σ_1, Σ_2 and Σ_3 are as in figure 1. Transition t_{12} of figure 4 arises by basic synchronisation from transitions t_1 and t_2 of $\Sigma_1\|\Sigma_2\|\Sigma_3$ as follows: t_1 and t_2 contain conjugate action terms $\overline{X}(a,b)$ and $X(c,d)$. To synchronise them by the operator $\mathbf{sy}\, X$, the variables in $Var(t_1)$ and $Var(t_2)$ are first made disjoint by a first renaming, and then the parameter lists of $\overline{X}(a,b)$ and $X(c,d)$ are considered equal by a second renaming. The effect is that a is considered the same variable as c and b is considered the same variable as d in the vicinities of t_1 and t_2 (but there are other suitable identifications – in general, introduction of fresh variables is necessary). Modulo these identifications, t_{12} is the multiset sum of t_1 and t_2 except that its guard is the logical conjunction of the guards of t_1 and t_2 and the two action terms $\overline{X}(a,b)$ and $X(c,d)$ which have effected the synchronisation are subtracted from the resulting label. Transition t_{13} of figure 4 is a synchronisation of transitions t_1 and t_3 of figure 1 according to the same principle. Note that it is internal.

Suppose that $X(v,w)$ and $Y(v',w')$ represent, respectively, a value change of variable X from v to w and of variable Y from v' to w'. Then the above example can be viewed as a high-level net representation of the program fragment

$$\cdots\ \mathbf{begin}\ \ \mathbf{var}\ \mathrm{X}{:}\{0,1\}\ (\mathbf{init}\ 0);\ \langle X := Y\rangle \parallel \langle X := 1{-}X\rangle\ \ \mathbf{end}\ \cdots$$

which specifies a block with a local variable X of type $\{0,1\}$ and a global variable Y which is assumed to be declared of type $\{0,1\}$ in some outer block. Nets Σ_1, Σ_2 and Σ_3 of figure 1 correspond, in that order, to the declaration $\mathbf{var}\,\mathrm{X}{:}\{0,1\}$, the atomic action $\langle X{:=}Y\rangle$ and the atomic action $\langle X{:=}1{-}X\rangle$. The label $\{X(c,d),Y(e,f)\}$ and the guard $d{=}e{=}f$ in Σ_2 indicate that the value of Y remains the same ($e{=}f$) while the value of X changes from whatever it was previously (c) to the value of Y ($d{=}e$). Thus, t_2 is a translation of $\langle X{:=}Y\rangle$. Similarly, t_3 is a translation of $\langle X{:=}1{-}X\rangle$. Transition t_1 represents all possible value changes of X.

The synchronisation and restriction $(\Sigma_1\|\Sigma_2\|\Sigma_3)\ \mathbf{sy}\, X\ \mathbf{rs}\, X$ describes the block in the sense that in the resulting net of figure 4, any value changes of X are described by appropriate connections to the place corresponding to X. This models the fact that X is local to the block. Any value changes of Y, however, are described implicitly by a label. This models the fact that Y is global to the block. Transition t_{12} is waiting to be synchronised with a corresponding transition carrying a conjugate label $\overline{Y}$ coming from the declaration of Y in an outer block.

The **sy** and **rs** operators enjoy a uniform set of coherence and algebraic properties [4]; both are coherent with respect to unfolding and commutative and idempotent with respect to action symbols:

$$\begin{array}{ll}
U(\Sigma\ \mathbf{rs}\, A) \quad\ = U(\Sigma)\ \mathbf{rs}\, A & \quad U(\Sigma\ \mathbf{sy}\, A) \quad\ = U(\Sigma)\ \mathbf{sy}\, A \\
(\Sigma\ \mathbf{rs}\, A)\ \mathbf{rs}\, B = (\Sigma\ \mathbf{rs}\, B)\ \mathbf{rs}\, A & \quad (\Sigma\ \mathbf{sy}\, A)\ \mathbf{sy}\, B = (\Sigma\ \mathbf{sy}\, B)\ \mathbf{sy}\, A \\
(\Sigma\ \mathbf{rs}\, A)\ \mathbf{rs}\, A = \Sigma\ \mathbf{rs}\, A & \quad (\Sigma\ \mathbf{sy}\, A)\ \mathbf{sy}\, A = \Sigma\ \mathbf{sy}\, A.
\end{array}$$

As a consequence, $\Sigma\,\mathbf{rs}\,\mathcal{A}$ and $\Sigma\,\mathbf{sy}\,\mathcal{A}$ are well-defined for any finite set $\mathcal{A}$ of action names. Furthermore, all operations are stable with respect to unfolding-equivalence. That is, if $\Sigma \equiv \Sigma'$ and $\Sigma_i \equiv \Sigma_i'$ (i=1, 2) then $\Sigma_1 \| \Sigma_2 \equiv \Sigma_1' \| \Sigma_2'$ as well as $\Sigma\,\mathbf{rs}\,A \equiv \Sigma'\,\mathbf{rs}\,A$ and $\Sigma\,\mathbf{sy}\,A \equiv \Sigma'\,\mathbf{sy}\,A$.

3 Behavioural notions

In section 3.1, occurrence sequences – i.e. interleavings – of M-nets are defined. Section 3.2 introduces a way of describing the collapsing of two sets of sequential transitions into single ones. Section 3.3 defines a notion of implementation between two (initially marked) M-nets and describes some examples.

3.1 Occurrence sequences and permutations

Consider an M-net Σ with initial marking $M^0{=}M^0_\Sigma$. An *occurrence sequence* of Σ starting with an arbitrary marking M_0 is defined as a (finite or infinite) alternating sequence

$$\tau \;=\; M_0(t_1,\sigma_1)M_1(t_2,\sigma_2)M_2\ldots$$

such that $(M_{j-1} \xrightarrow{t_j:\sigma_j} M_j)$ for $j{\geq}1$. A marking M is called *reachable* in Σ if there is an occurrence sequence from the initial marking M^0 to M.

We now define an equivalence on occurrence sequences with respect to permutations of independent transitions in the style of [2, 14]. For a transition t of Σ let ${}^\bullet t$ and $t^\bullet$ denote the sets of places $\{s|\iota(s,t){\neq}\emptyset\}$ and $\{s|\iota(t,s){\neq}\emptyset\}$, respectively. The transition sets ${}^\bullet s$ and $s^\bullet$ (for $s{\in}S$) are defined similarly.

Let t and t' be transitions of Σ (which could be the same ones) and let M be a marking of Σ. We define (t,t') to be *enabled concurrently* under a pair of bindings (σ,σ') of $Var(t)$ and $Var(t')$, respectively, if for all places s, the multiset of values $(\iota(s,t)[\sigma]{+}\iota(s,t')[\sigma'])$ is a sub-multiset of $M(s)$. If M enables t and t' concurrently under σ and σ', then M enables both t under σ and t' under σ'. However, the converse need not be true, because (t,σ) and (t',σ') may be in conflict under M. A sufficient condition for them to be concurrently enabled is that they are both enabled and ${}^\bullet t{\cap}{}^\bullet t'{=}\emptyset$.

Let τ and τ' be two occurrence sequences of Σ. We define $\tau \equiv_0 \tau'$ if τ and τ' can be decomposed as follows:

$$\tau \;=\; \tau_1\,\underbrace{(t,\sigma)M(t',\sigma')}_{(1)}\,\tau_2 \qquad\text{and}\qquad \tau' \;=\; \tau_1'\,\underbrace{(t',\sigma')M'(t,\sigma)}_{(2)}\,\tau_2'$$

such that $\tau_1{=}\tau_1'$, $\tau_2{=}\tau_2'$ and the last marking of τ_1 (which is also the last marking of τ_1') enables (t,σ) and (t',σ') concurrently. Note that not necessarily $M{=}M'$. For example, in figure 5 we have

$$M^0(t,\sigma_1)M^1(u,\sigma_2)M^2(t',\sigma_3)M^3 \;\equiv_0\; M^0(t,\sigma_1)M^1(t',\sigma_3)M^{2'}(u,\sigma_2)M^3$$

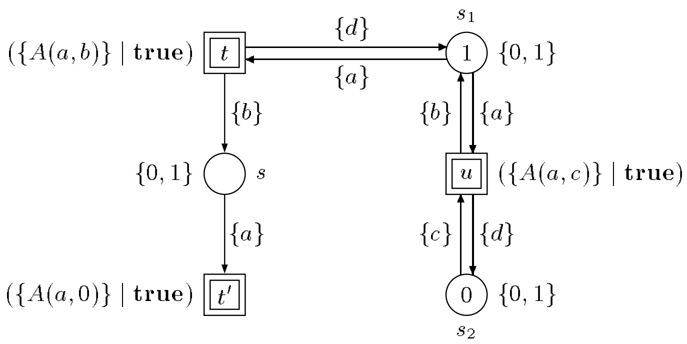


Figure 5. Illustration of $\equiv_0$ and of collapsing

with $\sigma_1 : a{\mapsto}1, b{\mapsto}1, d{\mapsto}1$, $\sigma_2 : a{\mapsto}1, b{\mapsto}1, c{\mapsto}0, d{\mapsto}0$ and $\sigma_3 : a{\mapsto}1$.

For finite τ and τ' we define $\tau \equiv \tau'$ iff $\tau \equiv_0^* \tau'$.[4] For a sequence τ, $[\tau]_\equiv$ denotes its equivalence class with respect to $\equiv$. Two $\equiv$-equivalent sequences can be viewed as expressing the same behaviour of Σ modulo commutativity of concurrently enabled transitions.

The consideration of $\equiv$-equivalence is necessary because, for example, a transition on the specification level may be split into two successive transitions on the implementation level, an opening one and a closing one (such as t and t' in figure 5). In an interleaving behaviour, an instance of an opening transition may occur much earlier than a corresponding instance of the closing transition. By $\equiv$-permutations they may be brought into neighbouring positions. This is described in the next section.

3.2 Collapsing transitions

Two nonempty disjoint sets of transition $T{\subseteq}T_\Sigma$ and $T'{\subseteq}T_\Sigma$ are called *in sequence* (in Σ) if there exists a place $s_{TT'}$ such that

- $M^0_\Sigma(s_{TT'}){=}\emptyset$,
- $\forall t'{\in}T'{:}\, {}^\bullet t'{=}\{s_{TT'}\}$ or $\forall t{\in}T{:}\, t^\bullet{=}\{s_{TT'}\}$,
- for all $t{\in}T$ and $t'{\in}T'$: $|\iota(t, s_{TT'})|{=}1{=}|\iota(s_{TT'}, t')|$.

The idea is that place $s_{TT'}$ sequentialises every transition in T with every transition in T'. This ensures – in a sense to be made precise – that when $t{\in}T$ is combined with $t'{\in}T'$, behavioural properties remain unchanged. For example, $\{t\}$ and $\{t'\}$ are in sequence in figure 5. Their sequentialising place is $s_{\{t\}\{t'\}}{=}s$.
A relation $\delta \subseteq 2^{T_\Sigma}{\times}2^{T_\Sigma}$ is called a *collapsing* if (a) whenever $(T,T'){\in}\delta$ then (T,T') are in sequence, and (b) the elements of δ are mutually disjoint, i.e., if $(T,T'){\in}\delta$ and $(U,U'){\in}\delta$, then $T{\cap}U{=}T{\cap}U'{=}T'{\cap}U{=}T'{\cap}U'{=}\emptyset$.[5] For instance, in figure 5, $\delta{=}\{(\{t\},\{t'\})\}$ is a collapsing.

[4] This definition can be generalised to infinite sequences by a more elaborate construction of forming the reflexive transitive closure; consult [2].
[5] That $T{\cap}T'{=}\emptyset$ and $U{\cap}U'{=}\emptyset$ already follows from their being in sequence.

Let a collapsing δ be given. An occurrence sequence τ is called δ*-complete* if, for every $(T,T')\in\delta$, whenever, in τ, some 'opening' instance of $t\in T$ puts some token m on place $s_{TT'}$ then there exists a subsequent 'closing' instance of some transition $t'\in T'$ which takes m away from $s_{TT'}$. For instance, in figure 5 with the δ defined above, $\tau_0{=}M^0(t,\sigma_1)M^1(u,\sigma_2)M^2$ is not δ-complete, but $\tau_1{=}M^0(t,\sigma_1)M^1(u,\sigma_2)M^2(t',\sigma_3)M^3$ is.

If τ is finite and δ-complete then, for any $(T,T')\in\delta$, $M(s_{TT'}){=}\emptyset$ in the marking M reached after τ. If $\delta{=}\emptyset$ then every occurrence sequence is δ-complete. By suitable $\equiv$-transformations, any δ-complete sequence can be brought into a canonical form: call τ δ*-respectful* if, whenever $(T,T')\in\delta$, every opening instance of $t\in T$ in τ, putting some token m on $s_{TT'}$, is immediately followed by a closing instance of some transition $t'\in T'$, removing this same token m from $s_{TT'}$.

Claim. Let τ be δ-complete. Then there exists a sequence $\tau'\in[\tau]_\equiv$ such that τ' is δ-respectful. ■3.2

The proof of this claim (which is given in the appendix) shows that there may be more than one candidates for τ'. We may pick any one of them and call it the canonical δ-respectful derivative, τ^δ, of τ, because all subsequent considerations are independent of this choice. If $\delta{=}\emptyset$ then $\tau^\delta{=}\tau$. For example, for the δ and τ_1 defined above (figure 5), we may choose $\tau_1^\delta{=}M^0(t,\sigma_1)M^1(t',\sigma_3)M^{2'}(u,\sigma_2)M^3$.

In the last part of this section we consider a marked M-net Σ, a collapsing δ on Σ, and a δ-respectful occurrence sequence τ of Σ. Our aim is to define an abstraction of τ called the δ*-trace* $\epsilon(\tau)$, by which we mean a sequence of (valuated) labels of transitions of τ. Intuitively, $\epsilon(\tau)$ describes the 'externally visible' behaviour generated by τ. We wish to consider neighbouring δ-related pairs of transitions as single entities; their combined label arises as the union of the two participating labels. We define $\epsilon(\tau)$ inductively. Suppose τ is of the form

$$\underbrace{\ldots\ M_{i-1}}_{(1)}(t_i,\sigma_i)M_i(t_{i+1},\sigma_{i+1})M_{i+1}(t_{i+2},\sigma_{i+2})\ \ \ldots$$

and suppose that $\epsilon(\tau)$ has already been constructed for the prefix up to and including (1). There are two possibilities for t_i: either t_i is in no set $T\in dom(\delta)$, or t_i is in some such set.

Consider the first possibility. Then we extend $\epsilon(\tau)$ by the multiset $\lambda(t_i)[\sigma_i]$, i.e. the multiset of action terms $\lambda(t_i)$ subjected to the binding σ_i (i.e. a multiset of ground action terms). If $\lambda(t_i)[\sigma_i]$ is empty then, by definition, this has the same effect as appending the empty word to $\epsilon(\tau)$, i.e. $\epsilon(\tau)$ is simply left as it was. The construction of $\epsilon(\tau)$ is then continued, starting with (t_{i+1},σ_{i+1}).

Now consider the second possibility. Then there is a pair $(T,T')\in\delta$ such that $t_i\in T$ and $t_{i+1}\in T'$. Valuation σ_i assigns values to $Var(t_i)$ including the (only) variable, say b, in $\iota(t_i,s_{TT'})$. Valuation σ_{i+1} assigns values to $Var(t_{i+1})$, including variable a in $\iota(s_{TT'},t_{i+1})$. Note that the value of b under σ_i equals the value of a under σ_{i+1} (since this is exactly the token put on place $s_{TT'}$ by t_i and removed from place $s_{TT'}$ by t_{i+1}). Without loss of generality, we may assume that the variables of t_i and t_{i+1} are disjoint. Then define $\sigma_i \cup \sigma_{i+1}$ as the binding of $Var(t_i) \uplus Var(t_{i+1})$ determined by σ_i and σ_{i+1}, and note that this is well-defined. Then, we extend $\epsilon(\tau)$ by the multiset $(\lambda(t_i)+\lambda(t_{i+1}))[\sigma_i \cup \sigma_{i+1}]$ of ground actions, and the construction of $\epsilon(\tau)$ is continued starting with (t_{i+2}, σ_{i+2}). For example, for τ_1^δ (figure 5) we have $\epsilon(\tau_1^\delta) = \{A(1,1), A(1,0)\}\{A(1,0)\}$.

Note that, in general, $\epsilon(\tau)$ is a sequence over the alphabet $\mathcal{G}(\Sigma, \delta)$ (the *valuated action symbols*), which is defined as follows: $\mathcal{G}(\Sigma,\delta) = \mathcal{G}^a \cup \mathcal{G}^b$, where $\mathcal{G}^a$ is the set of all $\lambda(t)[\sigma]$, with t being a transition of Σ *not* in some $T \cup T'$ with $(T, T') \in \delta$ and σ is a binding of $Var(t)$, and $\mathcal{G}^b$ is the set of all $(\lambda(t)+\lambda(t'))[\sigma \cup \sigma']$, where $t \in T$ and $t' \in T'$ with $(T, T') \in \delta$, σ and σ' are, respectively, bindings of $Var(t)$ and $Var(t')$, and $\sigma \cup \sigma'$ is as above.

3.3 Implementation relation

Let two marked M-nets Σ_1 and Σ_2, a collapsing δ_1 on Σ_1 and a collapsing δ_2 on Σ_2 be given. Let $\mathcal{G}_1 = \mathcal{G}(\Sigma_1, \delta_1)$ and $\mathcal{G}_2 = \mathcal{G}(\Sigma_2, \delta_2)$. Our aim here is to define under which circumstances Σ_1 may be called an 'implementation' of Σ_2. More precisely, we wish to capture the idea that every externally visible (δ_1-complete) behaviour of Σ_1 has a corresponding externally visible behaviour of Σ_2 (while the latter may have more behaviours which are not captured by the implementation).[6] Since there is no *a priori* relationship between the transition interfaces $\mathcal{G}_1$ and $\mathcal{G}_2$ of Σ_1 and Σ_2, this notion of implementation must be given with respect to a (partial) identification of those interfaces.

In this observational view, an *identification between Σ_1 and Σ_2* is a relation

$$\rho \subseteq \mathcal{G}_1 \times \mathcal{G}_2.$$

Relation ρ can be extended to a relation between sequences over $\mathcal{G}_1$ and sequences over $\mathcal{G}_2$: inductively, for finite sequences, ρ is the smallest relation such that $(\emptyset, \emptyset) \in \rho$ (as $\emptyset$ plays the role of empty sequence) and $(\epsilon_1, \epsilon_2) \in \rho$ for $\epsilon_1 \neq \emptyset \neq \epsilon_2$ iff

$$\exists\, \epsilon_1' \preceq \epsilon_1, \epsilon_2' \preceq \epsilon_2, g_1 \in \mathcal{G}_1, g_2 \in \mathcal{G}_2 \colon (\epsilon_1' g_1 = \epsilon_1) \wedge (\epsilon_2' g_2 = \epsilon_2) \wedge (\epsilon_1', \epsilon_2') \in \rho \wedge (g_1, g_2) \in \rho,$$

where $\preceq$ denotes the prefix relation; this definition can be extended canonically to infinite sequences.

In particular, for every abstracted sequence $\epsilon(\tau)$ of Σ_1 there may be a number of corresponding sequences ϵ' over $\mathcal{G}_2$ with $(\epsilon(\tau), \epsilon') \in \rho$. In general, such an ϵ' does not have to correspond to any occurrence sequence of Σ_2. If it always does, then we define Σ_1 to implement Σ_2. More precisely, let τ_1 be a δ_1-respectful occurrence sequence of Σ_1 and let τ_2 be a δ_2-respectful occurrence sequence of Σ_2. We define τ_1 and τ_2 to be *$\delta\rho$-related* if $(\epsilon(\tau_1), \epsilon(\tau_2)) \in \rho$. Moreover, Σ_1

[6] In terms of CCS this would mean, for instance, that a implements $a+b$.

is defined to *implement* Σ_2 *directly*, in symbols $\Sigma_1 \sqsubseteq_\rho^{\delta_1\delta_2} \Sigma_2$, iff for every δ_1-complete behaviour τ_1 of Σ_1 there exists a δ_2-complete behaviour τ_2 of Σ_2 such that $\tau_1^{\delta_1}$ and $\tau_2^{\delta_2}$ are $\delta\rho$-related.

In the remainder of this paper we always have $\delta_2{=}\emptyset$, and in the remainder of this section we abbreviate $\delta{=}\delta_1$ and $\sqsubseteq_\rho^\delta = \sqsubseteq_\rho^{\delta\emptyset}$.

Observe some special cases:

Claim. Assume Σ_2 is obtained from Σ_1 by omitting a place and its surrounding arcs, let $\delta{=}\emptyset$ and ρ identity. Then $\Sigma_1 \sqsubseteq_\rho^\delta \Sigma_2$.

Proof: Omitting a place at most adds, but does not restrict, behaviour. ∎3.3

Claim. Assume Σ_1 is obtained from Σ_2 by omitting a transition and its surrounding arcs, let $\delta{=}\emptyset$ and ρ the inclusion relation. Then $\Sigma_1 \sqsubseteq_\rho^\delta \Sigma_2$.

Proof: Omitting a transition at most restricts, but does not add, behaviour.
∎3.3

Moreover, let ρ be the identification between an M-net Σ and its unfolding $U(\Sigma)$, i.e. identity on valuated labels. Then $\Sigma \sqsubseteq_\rho^\emptyset U(\Sigma)$ and $U(\Sigma) \sqsubseteq_{\rho^{-1}}^\emptyset \Sigma$. Note also that $\sqsubseteq_\rho^\delta$ is stable over unfolding-preserving transformations, i.e. if $\Sigma_1'{\equiv}\Sigma_1$, $\Sigma_1 \sqsubseteq_\rho^\delta \Sigma_2$ and $\Sigma_2{\equiv}\Sigma_2'$, then $\Sigma_1' \sqsubseteq_{\rho'}^{\delta'} \Sigma_2'$ where ρ' and δ' are the same as ρ and δ except for the obvious identifications between Σ_1, Σ_1' and Σ_2, Σ_2'.

Note that $\sqsubseteq_\rho^\delta$ is a relation of 'direct implementation' which is not, in general, transitive, because collapsing a transition may be nested. Thus it is convenient to define its transitive closure. Let Σ_1 and Σ_2 (with their respective initial markings) and a relation ρ as above be given. Then Σ_1 is called an *implementation* of Σ_2 (with respect to ρ), in symbols $\Sigma_1 \sqsubseteq_\rho \Sigma_2$, if there are initially marked nets $\Sigma^0, \ldots, \Sigma^m$ and relations $\delta^1, \ldots, \delta^m$ as well as relations $\rho^1, \ldots, \rho^m$ such that

$$\Sigma_1{=}\Sigma^0\ ,\ \Sigma^m{=}\Sigma_2\ ,\ \Sigma^{j-1} \sqsubseteq_{\rho^j}^{\delta^j} \Sigma^j \text{ for } 1{\leq}j{\leq}m \text{ and } \rho = \rho^1 \circ \ldots \circ \rho^m .$$

This relation is obviously transitive.

The notion of implementation defined above is weak in the sense that, for instance, a net with an empty set of external traces would be called an implementation of any other net. In section 7.3 we discuss strengthening it.

Next in this section, we identify a class of pairs of M-nets of which the first implements the second. The idea is that whenever $t{\in}T$ and $t'{\in}T'$ for some $(T,T'){\in}\delta$ then, essentially, t and t' may be replaced by a single transition. We need this technique for the case that either: $t{\in}T$ is fixed and combined with every $t'{\in}T'$; or: every $t{\in}T$ is combined with a fixed $t'{\in}T'$; or: every $t{\in}T$ is combined with every $t'{\in}T'$.

Let Σ_1 be an initially marked M-net, let δ be a collapsing and let $(T,T'){\in}\delta$ with sequentialising place $s{=}s_{TT'}$. Moreover let $t_0{\in}T$. Let Σ_2 be the same net as Σ_1 except that: (a), for every pair (t_0,t') such that $t'{\in}T'$, a new transition, say

t_0t' (or $[t_0, t']$ to emphasise the fact that this new transition has been derived by collapsing from two old ones), with connections and inscription as defined below, is introduced; (b), transition t_0 is omitted from the net. We claim that Σ_1 implements Σ_2, and that in Σ_2, $T \backslash \{t_0\}$ and T' are in sequence.

The connections of t_0t' (i.e. its surrounding arcs) are the same as those of t_0 and t', minus the one that leads from t_0 to s and the one that leads from s to t'. Suppose the former has label $\{b\}$ and the latter has label $\{a\}$ in Σ_1 and suppose $Var(t_0) \cap Var(t') = \emptyset$. The guard of t_0t' is defined as $(\alpha(t_0)[b \leftarrow x] \wedge \alpha(t')[a \leftarrow x])$ where x is a fresh variable and $\alpha[y \leftarrow x]$ means α with y substituted by x. Finally, the label of t_0t' is defined as $(\lambda(t_0)[b \leftarrow x] + \lambda(t')[a \leftarrow x])$.

This construction is symmetric in the sense that every transition $t \in T$ could be paired with a fixed transition $t'_0 \in T'$, or every $t \in T$ could be paired with every $t' \in T'$, by exactly the same definition. Figure 6 shows an example of this transformation. The left-hand side shows a net Σ_1 with four transitions such that $\{t_1, t_2\}$ and $\{t'_1, t'_2\}$ are in sequence with a sequentialising place s. The right-hand side of the figure shows the net Σ_2 which results when both t_1 and t_2 are paired with transition t'_2.

Claim. Using the above notation (pairing a fixed $t \in T$ with all $t' \in T'$) let δ be $\{(\{t\}, T')\}$ and ρ be identity. Then $\Sigma_1 \sqsubseteq^{\delta}_{\rho} \Sigma_2$. ■3.3

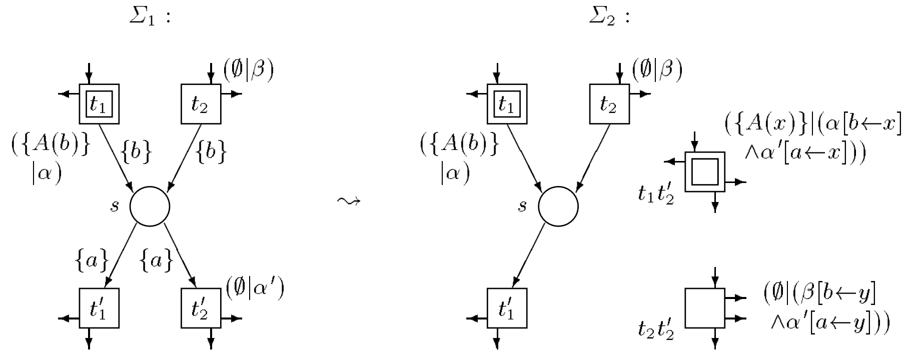


Figure 6. Example of (partial) collapsing of transitions

To end this section, we appeal to claim 3.3 to show that certain 'null loops' can be eliminated. To this end, consider figure 7. The M-net on its North-Western (NW) corner contains two places with the same value sets and a loop between them which shifts the value 1 from one place to the other. We show that these two places can, in effect, be combined into one place. The NE-net of figure 7 arises from the NW-net when place s is considered as a sequentialising place of its participating transitions.

The NW-net implements the NE-net (with the collapsing $\delta_1 = \{(\{z, w\}, \{y, u\})\}$ and identity $\rho_1 = \{(A, A)\}$) by claim 3.3. Furthermore, the NE-net implements the SW-net of figure 7 by a direct proof: every sequence of the former can be transformed into an equivalent sequence of the latter simply by omitting all instances of transition $[z, y]$ and one of the surrounding markings (which are equal because $[z, y]$ leaves markings invariant). Finally, the SW-net is unfolding-equivalent with the SE-net of figure 7 since transitions $[w, u]$ and $[w, y]$ have a **false** guard and can never be activated.

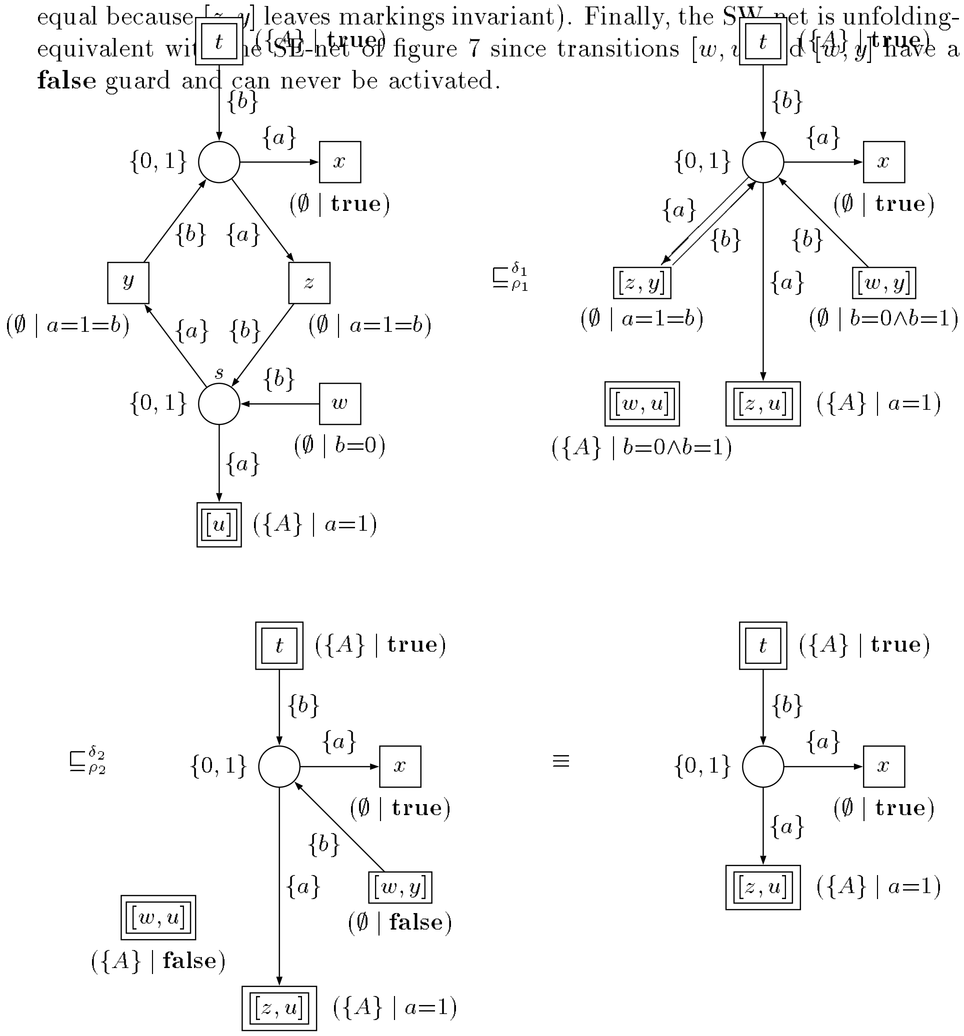


Figure 7. Illustration of the collapsing of 'null loops'

4 Memory specification

In this section we consider the specification problem defined in [6]. After giving some introductory definitions in section 4.1, we develop a specification *Spec* of the memory module by means of a sequence of increasingly complex M-nets in section 4.2 which capture the requirements of [6] one by one.

4.1 Notation

We use the following naming conventions:

$v, w \in Val$	(Values)
$l, l_0, \ldots \in Loc$	(Locations)
$p, p_0, \ldots \in Pcss$	(Processes)
$a, b, c, d, \ldots$	(Arc variables)
$ReadCall$, $WriteCall, \ldots$	(Action symbols).

We use arc variables in such a way that $\{a\}$ and $\{c\}$ always label arcs from places to transitions and $\{b\}$ and $\{d\}$ always label arcs from transitions to places. The RPC-memory specification problem calls for modules to be composed at the interfaces provided by their transitions. In the following we may therefore assume that all places are internal, i.e. have label **i**. We will simply omit these labels. Quotes from [6] will be reproduced in italic quotation style.

4.2 Reliable and unreliable memory

In this section we describe a series of specifications, $Spec0$–$Spec3$, such that $Spec0$ incorporates only the minimal requirements of a memory module while $Spec3$ incorporates the full set of requirements. For each of the specifications $Spec\mathbf{i}$, we also define two variants, $Spec\mathbf{i}'$ and $Spec\mathbf{i}u$ ($\mathbf{i}$=0, 1, 2, 3) which incorporate one type of exception and two types of exceptions, respectively: $Spec\mathbf{i}'$ is $Spec\mathbf{i}$ plus $BadArg$ exceptions and $Spec\mathbf{i}u$ is $Spec\mathbf{i}'$ plus $MemFail$ exceptions. Eventually, we will be interested in $Spec3'$ (which is 'reliable memory' according to [6]) and $Spec3u$ (which is 'unreliable memory' according to [6]).

A basic component, $Spec0$, is shown in figure 8. Procedure names and their parameters referring to the RPC-memory specification problem are expressed by action symbols and arc variables, respectively. The idea is that 'calling' the $\{ReadCall(p, l)\}$ transition of $Spec0$ consists of synchronising transition t_1 with some transition of the environment of $Spec0$ which contains a conjugate label $\{\overline{ReadCall}(p, l)\}$. Similarly, 'returning' a value after a *read* consists of synchronising the $\{ReadReturn(p, l, v)\}$ transition t_2 with some environment transition containing the conjugate label $\{\overline{ReadReturn}(p, l, v)\}$.

All transitions have a parameter $p \in Pcss$, which is a process identifier. This implements the following requirement from [6]:

> *'A component may contain multiple* processes *that can concurrently issue procedure calls. More precisely, after one process issues a call, other processes can issue calls to the same component before the component issues a return from the first call. A return action communicates to the calling component the identity of the process that issued the corresponding call'.*

For example, it may be the case that there are two processes p_1 and p_2 issuing $ReadCall$s one after the other. When process p_1 has issued a $ReadCall(p_1, l_1)$

to location l_1, place s^0 contains a token (p_1, l_1) and when process p_2 issues a $ReadCall(p_2, l_2)$ before p_1 has accepted the corresponding $ReadReturn(p_1, l_1, .)$, then place s^0 contains two tokens (p_1, l_1) and (p_2, l_2). It may even happen that $p_1{=}p_2$ and $l_1{=}l_2$ (then the marking on s^0 is a true multiset, allowing for a large degree of concurrency in general); it may also happen that $ReadReturn(p_2, l_2, .)$ is executed before $ReadReturn(p_1, l_1, .)$.

*Spec*0 merely specifies that every *ReadReturn* must be preceded by a *ReadCall* with the same process and location parameters and that every *WriteReturn* must be preceded by a *WriteCall* with corresponding parameters. This excludes 'spontaneous' read or write returns – a requirement left implicit in [6]. However, *Spec*0 does not implement

> *'The memory must eventually issue a return for every* Read *and* Write *call'.*

As this requirement is a typical progress assumption, it cannot be modelled by an ordinary net (nor by an ordinary program). Progress assumptions are captured by the consideration of completeness of executions as described in section 5.4. Note that in *Spec*0, reading and writing are oblique with respect to their parameter v. *Spec*0 imposes no restriction on value correctness. For instance, a value 5 may be written to a location l, followed immediately by a read of 3 from the same location l.

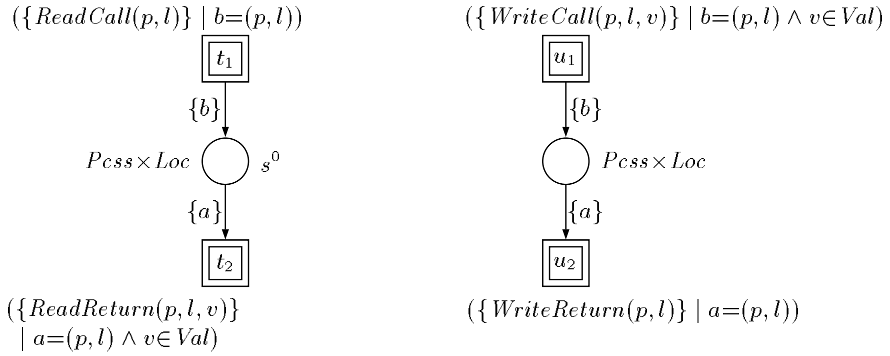


Figure 8. Specification *Spec*0

Before adding value correctness, we show how the left-hand side of *Spec*0 can be modified to yield two versions of 'less reliable' memory interface specification, *Spec*$0'$ and *Spec*$0u$, which allow exceptions as defined in [6] to happen. To express the requirement

> *'*BadArg*:argument* loc *is not an element of* MemLocs*',*

the existence of a set of non-locations called *NonLoc* is assumed. A $BadArg_r$ exception (transition v_1 in figure 9) is raised in case $l{\in}NonLoc$. The *MemFail* exception (transition v_2 in figure 9) models

*'*MemFailure*:the memory cannot be read'.*

Figure 9 shows the *read* part of *Spec*0*u*. We do not show the *write* part since it is analogous (it is not entirely symmetrical but easy to derive, and it will be shown later explicitly for *Spec*3*u*). We define *Spec*0′ as *Spec*0*u* minus transition v_2 and the corresponding *MemFail* transition in the *write* part (such that *Spec*0′ is the same as *Spec*0 plus *BadArg* exceptions).

Figure 9 illustrates a general pattern for adding exceptions to specifications. They are simply described as additional behavioral possibilities at the control flow places at which the exception may occur.

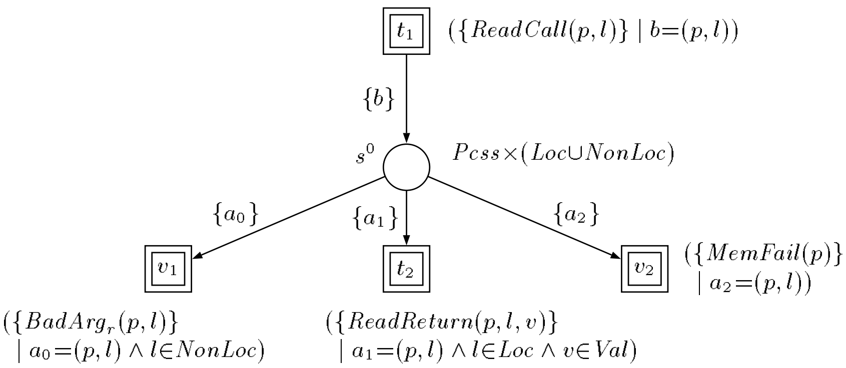


Figure 9. *Spec*0*u* – adding exceptions to *Spec*0 (only the *read* part)

Adding value correctness to *Spec*0 is done in *Spec*1 (figure 10) whose central place s models memory. The initial marking $\otimes$ of s is defined to be the set of pairs $\{(l, v_l^0)|l{\in}Loc\}$, where v_l^0 denotes the initial value at location l. This models

> *'The memory behaves as if it maintains an array of atomically read and written locations that initially all contain the value* InitVal*'.*

Note that s is connected by side conditions (only) to transitions t_2 and u_1 (which are the ones involving actual memory accesses). Transition u_1 specifies that a *WriteCall* fetches a value w from the specified memory location l (by $c{=}(l,w)$), and overwrites it by v, the parameter of the call, at the same location l (by $d{=}(l,v)$); by a *ReadReturn* (transition t_2), a value v is similarly fetched from the specified memory location l (by $c{=}(l,v)$) and returned to the caller (since v is a parameter of $ReadReturn(p,l,v)$) as well as to memory (since $d{=}(l,v)$). Note that the conjuncts $v{\in}Val$ of *Spec*0 have disappeared. They are now implicit because variables c and d can be bound only to tuples in $Loc{\times}Val$.

Exceptions may be added to *Spec*1 as described in the transformation from *Spec*0 into *Spec*0′ (adding *BadArg*) and further into *Spec*0*u* (adding *BadArg* and *MemFail*). The new transitions have no connection to place s, which models

'An operation that raises a BadArg *exception has no effect on the memory',*

and (for the time being) also that *MemFail* exceptions have no effect on the memory. Specifications *Spec*$1'$ and *Spec*$1u$ are not shown pictorially.

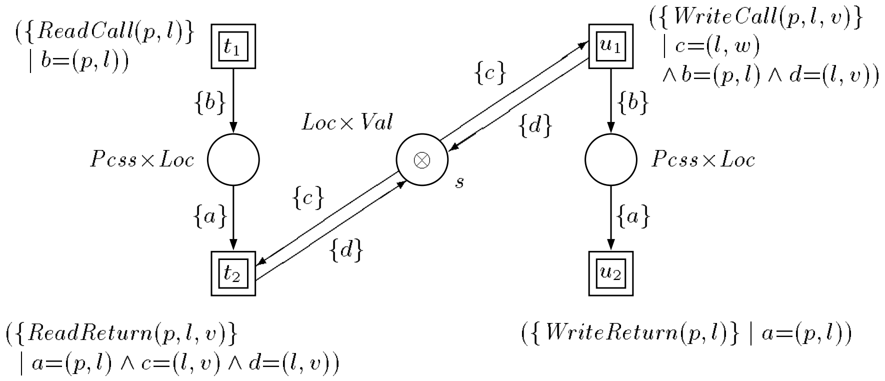


Figure 10. *Spec*1 – adding value correctness to *Spec*0 (by place s)

In the next step, which is specification *Spec*2 displayed in figure 11, we model read/write-detachedness, i.e. the 'at some time between' clauses of

'Each successful Read(l) *operation performs ... an atomic read to location* l *at some time between the call and return. Each successful* Write$(l,\ v)$ *operation performs ... atomic writes of value* v *to location* l *at some time between the call and return',*

by separating actual reading from *ReadReturn* and actual writing from *WriteCall*. In *Spec*2 the *ReadReturn* and *WriteCall* transitions, t_2 and u_1, of *Spec*1 are, therefore, split into two successive transitions, t_2', t_2'' and u_1', u_1''. Actual memory accesses are now internal rather than external. They cannot be invoked from the outside (i.e. cannot be synchronised with any matching conjugate transitions).

Figure 12 shows an unreliable version, *Spec*$2u$, of *Spec*2. We show only the general pattern of this specification because of its analogy to *Spec*$0u$ shown in figure 9. Note that as required [6], the $BadArg_r$ exception gets detected straight away before an actual read while *MemFail* can occur at any time. This models

'An operation that raises a BadArg *exception has no effect on the memory'.*

A *MemFail* exception may now have an effect on the memory if it is raised between a memory access and the corresponding (*write*) return.

In the final step of the specification we add the requirement that writes may be repeated [6]. This models the emphasised part of the sentence

'Each successful Write$(l,\ v)$ *operation performs* a sequence of one or more atomic writes *of value* v *to location* l *at some time between the call and return.'*

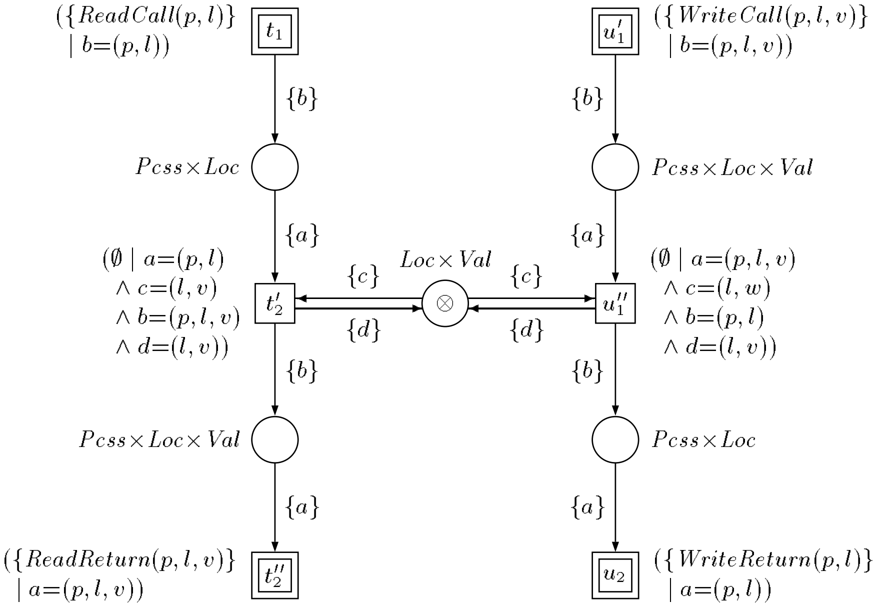


Figure 11. *Spec*2 – adding detachedness to *Spec*1 (splitting t_2 and u_1)

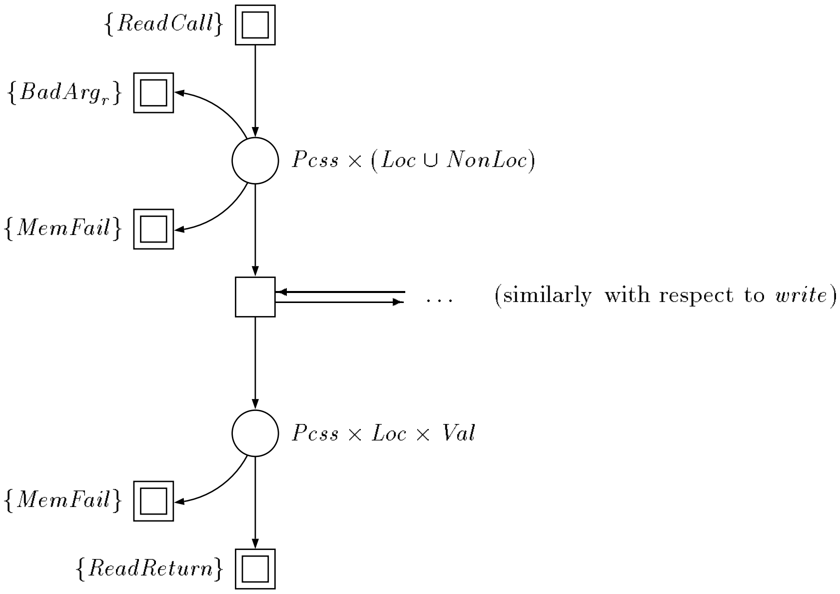


Figure 12. *Spec*2*u* – adding exceptions to *Spec*2 (pattern)

Again there are three versions of this specification: *Spec*3 (without exceptions), *Spec*3′ (with *BadArg* but no *MemFail* exceptions) and *Spec*3*u* (with *BadArg* and *MemFail* exceptions). This time we do not show *Spec*3, but we show *Spec*3′ (in figure 13) and *Spec*3*u* (in figure 17),[7] because they correspond, respectively, to

'reliable memory' and *'unreliable memory'*.

*Spec*3′ contains one additional transition, $\widetilde{u_1''}$, with respect to *Spec*2′, but note that in order to enable the repeated write, the place s' just below 'actual write' needs to have type $Pcss \times Loc \times Val$ instead of $Pcss \times Loc$ as in *Spec*2′. This allows transitions 'actual write' and 'repeated actual write' to put the value v with which they are called not only on s but also on s', which is necessary because when writes are repeated, the value which is to be written must still be known. Note that the inscription of the new transition is the same as that of the 'actual write'. The specification of unreliable memory with repeated writes, *Spec*3*u*, which is shown below in figure 17, can be obtained from *Spec*3′ in the same way as *Spec***i***u* is obtained from *Spec***i**′ (**i**=0, 1, 2), i.e. by adding *MemFail*.

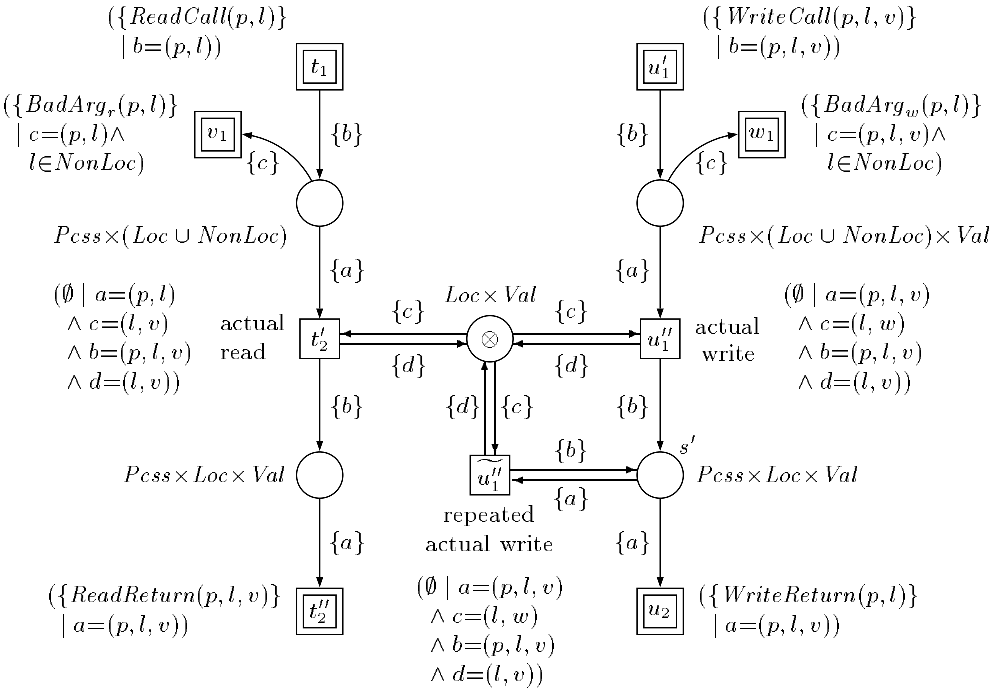


Figure 13. Reliable memory: *Spec*3′ – adding repeated write to *Spec*2′

For the purpose of the next claim let ρ be the inclusion relation and let $\sqsubseteq$

[7] This figure has been given a place rather later in the paper in order to enable a direct comparison between *Spec*3*u* and a net defined at a later point, *Impl*.

denote $\sqsubseteq_\rho^\emptyset$. Note that the identifications expressed by this choice of ρ are not freely invented but are part of the motivation behind the series of specifications.

Claim.
$$\begin{array}{ccccccc}
Spec0 & \sqsupseteq & Spec1 & \sqsupseteq & Spec2 & \sqsubseteq & Spec3 \\
\sqcap\!| & & \sqcap\!| & & \sqcap\!| & & \sqcap\!| \\
Spec0' & \sqsupseteq & Spec1' & \sqsupseteq & Spec2' & \sqsubseteq & Spec3' \\
\sqcap\!| & & \sqcap\!| & & \sqcap\!| & & \sqcap\!| \\
Spec0u & \sqsupseteq & Spec1u & \sqsupseteq & Spec2u & \sqsubseteq & Spec3u.
\end{array}$$
■4.2

Note that the series of specifications defined in this section does not provide a strictly increasing sequence of implementations. This lies in the nature of the problem: while value correctness restricts behaviour and detachedness leaves behaviour invariant modulo transition splitting, repeated write adds behaviour. Moreover, $Spec3u$ does not implement $Spec3'$ with respect to the choice of ρ, because $Spec3u$ has an execution $M^0\{ReadCall\}M^1\{MemFail\}M^2$ for which no corresponding execution exists in $Spec3'$. We select $Spec{=}Spec3u$, i.e. unreliable remote memory with repeated writes, to be our specification of interest. An implementation of *Spec*, using a remote procedure call interface and reliable memory $Spec3'$, is described in the next section.

5 Memory implementation

In section 5.1 we specify a Remote Procedure Call (*RPC*) interface, by means of which a user may issue remote calls and receive remote returns using the specific parametric 'procedure constants' *read* and *write*. Section 5.2 describes an implementation of *Spec* using an *RPC* interface with access to reliable memory as specified by $Spec3'$. Section 5.3 describes the main result. Finally, section 5.4 investigates progress and fairness properties of the solution.

5.1 The *RPC* module

We introduce new naming conventions:

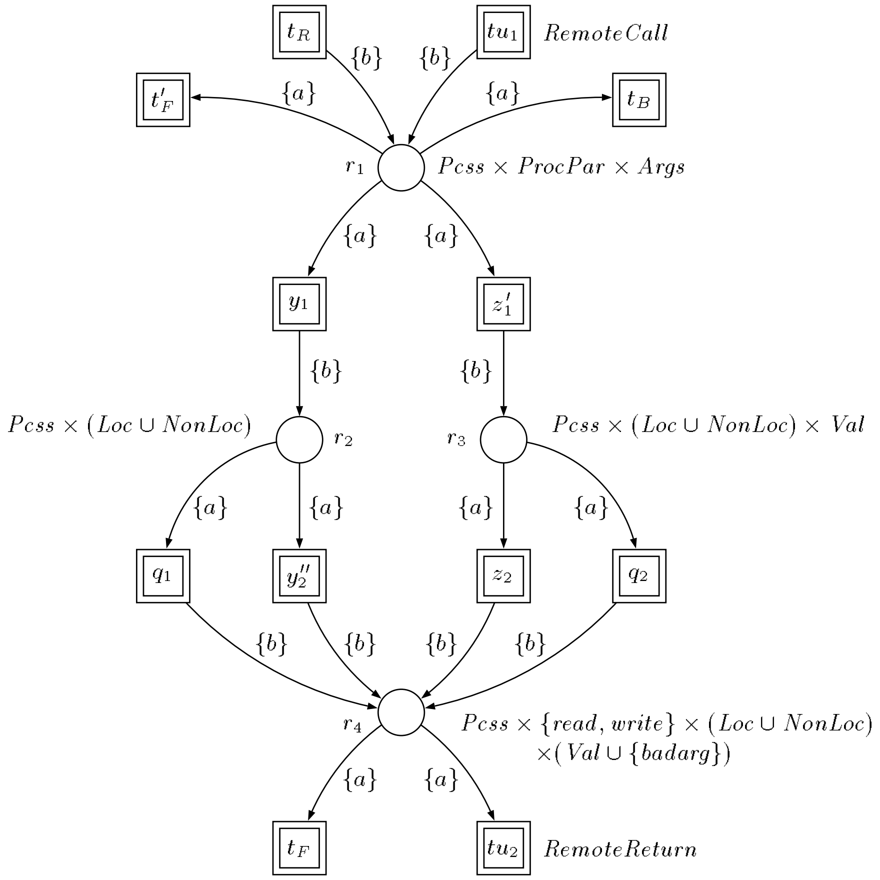
$$\begin{array}{rl}
r \in \{read, write\} \subseteq ProcPar & \text{(Procedure parameters)} \\
arglist = [arg_1, \ldots, arg_k] \in Args & \text{(Argument lists)}
\end{array}$$

with $|arglist|{=}k$, $ArgNum(read){=}1$ and $ArgNum(write){=}2$. This models

> '... *mapping* ArgNum*, where* ArgNum(p) *is the number of arguments of each procedure p'*

for the two procedure constants, *read* and *write*, which are relevant in our context.[8] Figure 14 shows the specification of the *RPC* interface. When *RPC* is called in mode *read*, it either has parameter list $arglist{=}[l]$ (with a location l) or raises an exception, and when *RPC* is called in mode *write*, it has parameter list $arglist{=}[l, v]$ (with a location l and value v) or raises an exception. Conversely, when *RPC* returns in mode *read*, it has parameter list $arglist{=}[l, v]$ and when it returns in mode *write*, it has parameter list $arglist{=}[l]$.

[8] What has been called p in [6] has generally been called r in this paper.



$$
\begin{aligned}
\iota(tu_1) &= (\,\{RemoteCall(p, r, arglist)\} \mid b{=}(p, r, arglist)\,)\\
\iota(y_1) &= (\,\{\overline{ReadCall}(p, l)\} \mid a{=}(p, read, [l]) \wedge b{=}(p, l)\,)\\
\iota(z'_1) &= (\,\{\overline{WriteCall}(p, l, v)\} \mid a{=}(p, write, [l, v]) \wedge b{=}(p, l, v)\,)\\
\iota(t_B) &= (\,\{BadCall(p)\} \mid a{=}(p, r, arglist) \wedge BADCALL(p, r, arglist)\,)\\
\iota(t'_F) &= (\,\{RPCFailure(p, r, arglist)\} \mid a{=}(p, r, arglist)\,)\\
\iota(t_F) &= (\,\{RPCFailure(p, r, arglist)\} \mid\\
&\quad (a{=}(p, read, l, v) \wedge arglist{=}[l]) \vee (a{=}(p, write, l, v) \wedge arglist{=}[l, v])\,)\\
\iota(t_R) &= (\,\{RPCRetry(p, r, arglist)\} \mid b{=}(p, r, arglist)\,)\\
\iota(y''_2) &= (\,\{\overline{ReadReturn}(p, l, v)\} \mid a{=}(p, l) \wedge b{=}(p, read, l, v)\,)\\
\iota(z_2) &= (\,\{\overline{WriteReturn}(p, l)\} \mid a{=}(p, l, v) \wedge b{=}(p, write, l, v)\,)\\
\iota(tu_2) &= (\,\{RemoteReturn(p, r, arglist)\} \mid a{=}(p, r, l, v) \wedge\\
&\quad (\,(v{\in} Val \wedge r{=}read \wedge arglist{=}[l, v]) \vee (v{\in} Val \wedge r{=}write \wedge arglist{=}[l])\\
&\quad \vee (v{=}badarg \wedge arglist{=}[l, badarg])\,)\,)\\
\iota(q_1) &= (\,\{\overline{BadArg_r}(p, l)\} \mid a{=}(p, l) \wedge b{=}(p, read, l, badarg) \wedge l{\in} NonLoc\,)\\
\iota(q_2) &= (\,\{\overline{BadArg_w}(p, l)\} \mid a{=}(p, l, v) \wedge b{=}(p, write, l, badarg) \wedge l{\in} NonLoc\,).
\end{aligned}
$$

Figure 14. Specification of the RPC component

By its transitions tu_1 and tu_2, the *RPC* interface interacts with any user (a 'sender' [6]) who is capable of issuing $\overline{RemoteCall}(\ldots)$ commands with appropriate parameter lists and using $\overline{RemoteReturn}(\ldots)$ commands to wait for the results of previous calls. By its transitions y_1, z_1', y_2'', z_2, q_1 and q_2, the *RPC* module interacts with any of the specifications *Spec***i**, *Spec***i**$'$ or *Spec***i***u* (**i**$=0,1,2,3$) defined in the previous section (the 'receivers' [6]). With respect to *Spec***i**, *Spec***i**$'$ and *Spec***i***u*, the *RPC* interface plays the role of a caller (made explicit by over-barring action names) while *Spec***i**, *Spec***i**$'$ or *Spec***i***u* play the role of callees. More precisely, depending on the parameter r with which it is called, the *RPC* module invokes either *ReadCall* or *WriteCall* and waits for the subsequent *Return*s or *BadArg* exceptions. This models

> *'The RPC component interfaces with two environment components, a* sender *and a* receiver*. It relays procedure calls from the sender to the receiver, and relays the return values back to the sender'.*

RPC may also raise *BadCall* and *RPCFailure* exceptions (cf. transitions t_B, t_F' and t_F) to the sender. Moreover, *RPC* has a retry transition t_R which is simply a differently labelled copy of the *RemoteCall* transition tu_1. This allows t_R to be synchronised without tu_1 being synchronised at the same time.

It is not exactly clear from [6] under which circumstances exception *BadCall* should be raised. The two relevant sentences are

> *'*BadCall*:*proc *is not a valid name or* args *is not a syntactically correct list of arguments for* proc*';*
> *'Raise a* BadCall *exception if* args *is not a list of* ArgNum(proc) *arguments'.*

Hence the transition t_B labelled *BadCall* in figure 14, which models this, has a parametric guard $BADCALL(p, r, arglist)$, which could be either

$$r \notin \{read, write\} \vee \neg(\text{good-syntax}),$$

to formalise the notion of 'syntactically correct', or $|arglist| \neq ArgNum(r)$ to formalise the notion of 'the argument list does not have the right number of elements'.

The *RPC* component detects any *BadArg* exceptions raised by either *ReadCall* (transition q_1 in figure 14) or *WriteCall* (transition q_2 in figure 14). Upon detection of such an exception, *RPC* passes it on to the calling environment in the form of a special value $badarg \notin Val$ which is accommodated in the type of place r_4. This models

> *'Raises any exception raised by a call to* proc*'.*

Normal return is indicated by tu_2 being executed in mode $r{=}read$ or $r{=}write$, while tu_2 being executed in mode $v{=}badarg$ indicates exception propagation of *BadArg* to the sender. Transition tu_2 thus models the (a) part of

'Issue one call to procedure proc *with arguments* args, *wait for the corresponding return (which the RPC component assumes will occur) and either* (a) *return the value (normal or exceptional) returned by that call, or* (b) *raise the* RPCFailure *exception',*

Moreover, transition t_F models the (b) part of the sentence quoted above, while transition t'_F models

'Issue no procedure call, and raise the RPCFailure *exception'.*

The requirement

'... (which the RPC component assumes will occur) ...',

is modelled by places r_2 and r_3 not having any attached *RPCFailure* exceptions and by considering progress/timing assumptions (cf. sections 5.4 and 6).

5.2 The implementation

Before putting *RPC* and reliable memory *Spec3'* in parallel and synchronising them, the requirement

'If the call (to RPC) raises an RPCFailure *exception, then the implementation may either reissue the call to the RPC component or raise a* MemFailure *exception'*

needs to be dealt with. This specifies a retry facility which, besides invoking a fresh *RemoteCall* after an *RPCFailure*, may result in a memory failure *RPCMemFail* at the *RPC* module level.

For this purpose, the *RPC* module needs to have the further interface transition t_R. Even after a successful *WriteReturn*, the *RPC* module cannot afford to ignore the value v of an attempted *write*, unless a corresponding *RemoteReturn* has actually been executed. This is because in case of an *RPCFailure*, the retry module must be aware of the value that should have been written. Thus, tokens on place r_4 of figure 14 have to carry a value even if they correspond to a past *write*. The value v of such a *write* is dropped only when a normal return happens, i.e. through the annotation of transition *RemoteReturn*.

The retry behaviour specified above is modelled by the *Retry* module shown in figure 15. Note how the overbarring in figure 15 provides for the interaction of this module with its environment: overbarred transitions labelled $\overline{RPCFailure}$ and $\overline{RPCRetry}$ (i.e. t'_R) denote interactions between *Retry* and transitions t'_F, t_F and t_R of *RPC* (figure 14), while non-overbarred transition t_M (*RPCMemFail*) signals an exception to the environment at the same level as the *RemoteCall*, *BadCall* and *RemoteReturn* transitions of *RPC*.

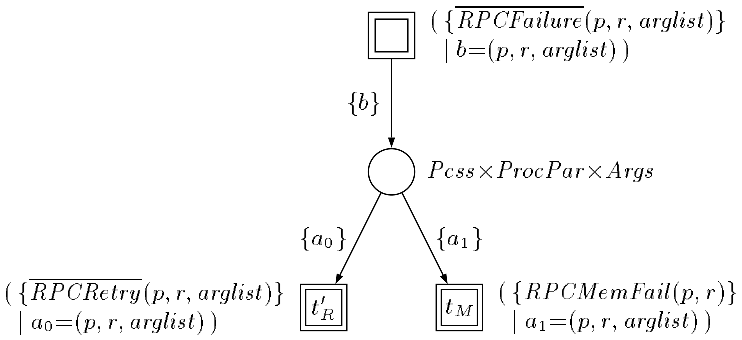


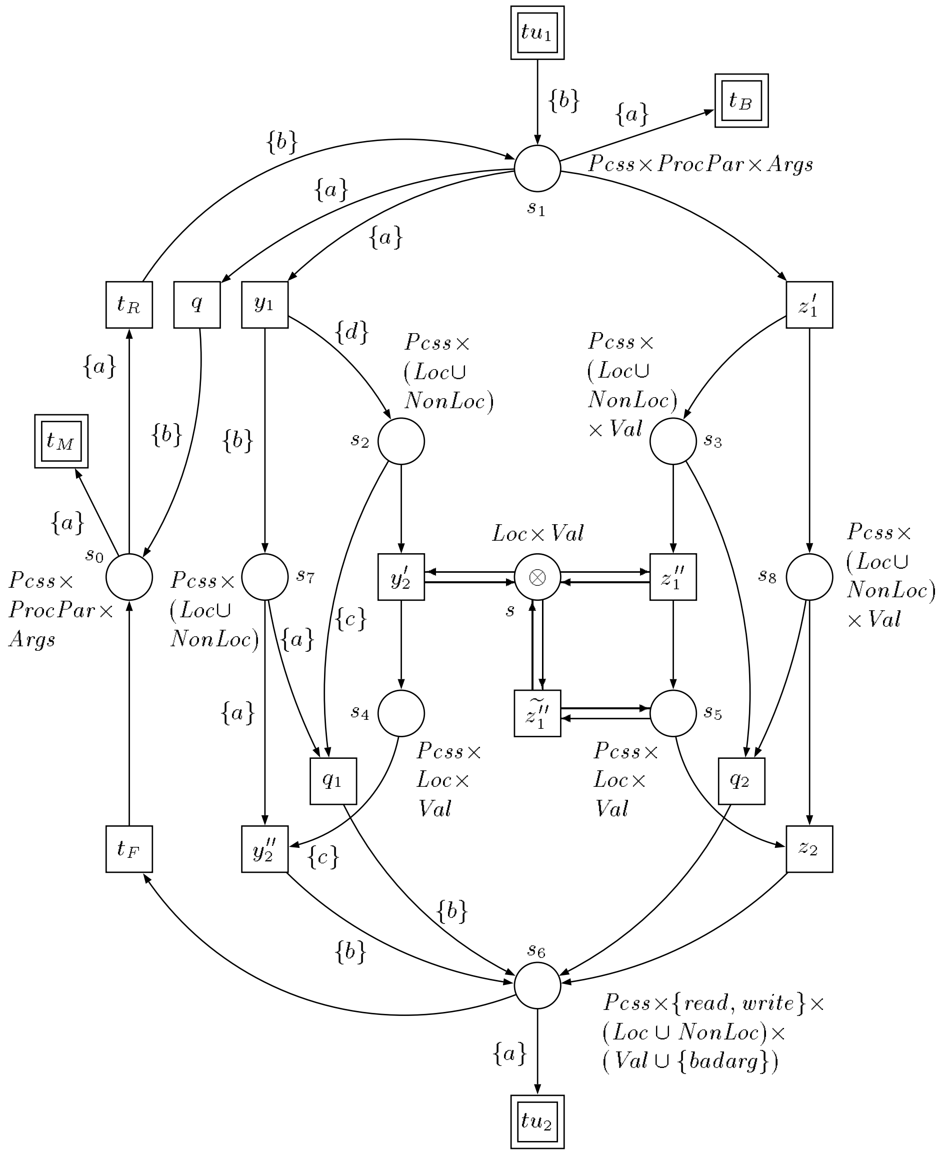
Figure 15. The *Retry* module

We may now put together the *RPC* module, the *Retry* module and reliable memory *Spec*3′ as follows:

$$\begin{array}{lcl} Impl & \equiv & (\,(Retry \parallel RPC \parallel Spec3')\\ & & \;\mathbf{sy}\ \{\ ReadCall, ReadReturn, WriteCall, WriteReturn,\\ & & \qquad RPCFailure, RPCRetry, BadArg_r, BadArg_w\ \}\\ & & \;\mathbf{rs}\ \{\ ReadCall, ReadReturn, WriteCall, WriteReturn,\\ & & \qquad RPCFailure, RPCRetry, BadArg_r, BadArg_w\ \}\,). \end{array}$$

Note that according to section 2.3, this is a well-defined M-net. It is shown in figure 16. We briefly explain the origins of the transitions of *Impl*. The internal transition t_R comes from a synchronisation of transitions t_R of figure 14 and t'_R of figure 15 over the pair $\{RPCRetry, \overline{RPCRetry}\}$ of conjugate action symbols.[9] Transition y_1 comes from synchronising transitions labelled $\overline{ReadCall}$ (in *RPC*, figure 14) and *ReadCall* (in *Spec*3′, figure 13). Transitions y''_2, z'_1 and z_2 arise from similar synchronisations. Transition t_B signals the *BadCall* exception of the *RPC* component to its environment (at the same level as the *RPCMemFail* exception t_M). Transition q stems from synchronising the *RPCFailure* transition t'_F of *RPC* (figure 14) with the $\overline{RPCFailure}$ transition of *Retry*. Transition t_F comes similarly from synchronising the *RPCFailure* transition t_F of *RPC* with the $\overline{RPCFailure}$ transition of *Retry*. Transition q_1 comes from synchronising the $BadArg_r$ transition of *Spec*3′ with the $\overline{BadArg_r}$ transition q_1 of *RPC*; it describes the detection of the *BadArg* exception after a *ReadCall*. Transition q_2 describes similarly the detection of a *BadArg* exception after a *WriteCall*. Note that q, t_F, q_1 and q_2 are internal to the implementation. The detection of *BadArg* exceptions is passed to the environment by creating the special token *badarg* on place s_6, which is a copy of r_4.

[9] We strive to use names for transitions and places consistently. However, sometimes it is convenient to re-use already used names, such as t_R in figures 14 and 16.



$$
\begin{aligned}
\iota(tu_1) &= (\,\{RemoteCall(p,r,arglist)\} \mid b{=}(p,r,arglist)\,)\\
\iota(t_B) &= (\,\{BadCall(p)\} \mid a{=}(p,r,arglist) \wedge BADCALL(p,r,arglist)\,)\\
\iota(y_1) &= (\,\emptyset \mid a{=}(p,read,[l]) \wedge b{=}(p,l) \wedge d{=}(p,l)\,)\\
\iota(q) &= (\,\emptyset \mid a{=}(p,r,arglist){=}b\,) \quad (= \;\; \iota(t_R)\,)\\
\iota(t_M) &= (\,\{RPCMemFail(p,r)\} \mid a{=}(p,r,arglist)\,)\\
\iota(q_1) &= (\,\emptyset \mid a{=}(p,l){=}c \wedge b{=}(p,read,l,badarg) \wedge l{\in}NonLoc\,)\\
\iota(y_2'') &= (\,\emptyset \mid a{=}(p,l) \wedge c{=}(p,l,v) \wedge b{=}(p,read,l,v)\,)\\
\iota(tu_2) &= (\,\{RemoteReturn(p,r,arglist)\} \mid a{=}(p,r,l,v)\wedge\\
&\quad (\,(v{\in}Val{\wedge}r{=}read{\wedge}arglist{=}[l,v]) \vee (v{\in}Val{\wedge}r{=}write{\wedge}arglist{=}[l])\\
&\quad \vee(v{=}badarg{\wedge}arglist{=}[l,badarg])\,)\,).
\end{aligned}
$$

Figure 16. Implementation *Impl* and some of its inscriptions

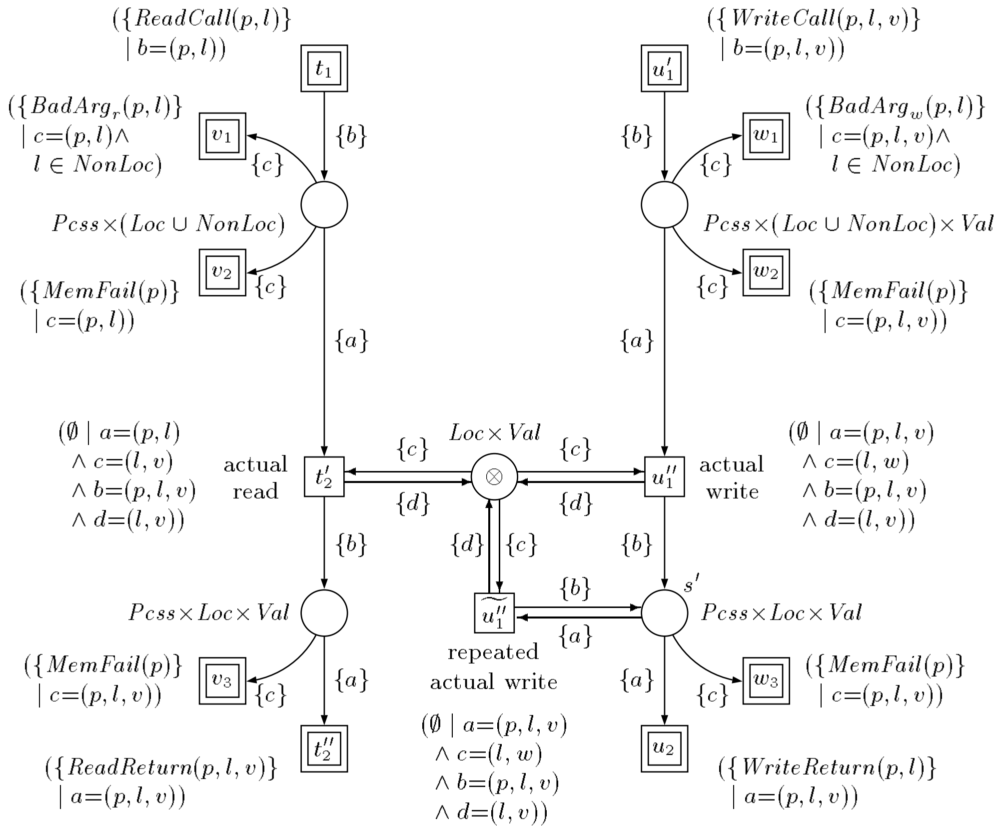


Figure 17. Unreliable memory: Specification $Spec = Spec3u$

5.3 Main result

Figure 17 shows the specification *Spec*=*Spec*3*u*. To compare *Impl* and *Spec*, problem-inherent identifications are formalised by a relation

$$\rho_{Impl} \subseteq \mathcal{G}_1 \times \mathcal{G}_2,$$

where $\mathcal{G}_1$ are the valuated action symbols of *Impl* (figure 16) and $\mathcal{G}_2$ are the valuated action symbols of *Spec* (figure 17); see table 1. This relation is given implicitly in [6], but there is room for a few arbitrary choices. Firstly, if the *BadCall* exception is raised in *Impl*, then this could be mapped either to the *MemFail* exceptions or to the *BadArg* exceptions of *Spec* (or indeed to both of them). For the sake of simplicity, we have chosen to map any *BadCall* to *MemFail*. This justifies lines (e) and (f) of table 1. Secondly, if a *RemoteCall* is executed with a syntactically incorrect parameter list, then this must eventually yield a *BadCall* exception in *Impl* and is modelled by a *MemFail* exception in *Spec*, but before the latter, a *ReadCall* or a *WriteCall* must occur (depending on the mode in which *RemoteCall* was executed). To enable such a call, we need to assume that both *Loc* and *Val* are nonempty sets, so that we may pick arbitrary (but fixed) elements $l_0 \in Loc$ and $v_0 \in Val$. This justifies lines (c) and (d) of table 1, where it is assumed that $arglist_0 \neq [l]$ and $arglist'_0 \neq [l,v]$. All other elements of ρ_{Impl} are determined by the problem statement.

To understand table 1, consider, for instance, line (i). Check figure 16 to see that in mode *read*, the parameter list of tu_2 is $(p, read, [l,v])$, and check figure 17 to see that the parameter list of t''_2 is (p,l,v). Consider a binding with $p{=}p_1$ (a concrete procedure p_1), $l{=}l_1$ (a concrete location l_1) and $v{=}5$ (a concrete value). Line (i) of table 1 means that the valuated label $\{RemoteReturn(p_1, read, [l_1, 5])\}$ of *Impl* is, by definition, related to the valuated label $\{ReadReturn(p_1, l_1, 5)\}$ of *Spec*, i.e. $(\{RemoteReturn(p_1, read, [l_1, 5])\}, \{ReadReturn(p_1, l_1, 5)\}) \in \rho_{Impl}$.

The other lines of table 1 have similar meaning, and the main result is:

Claim. *Impl* implements *Spec* w.r.t. ρ_{Impl}, i.e. $Impl \sqsubseteq_{\rho_{Impl}} Spec$. ■5.3

5.4 Progress and fairness

In this section we investigate the progress and fairness properties of *Impl* in dependence of the same properties with respect to its constituents, *Retry*, *RPC* and *Spec*3′. To start with, we define these notions. Let a marked M-net Σ and an occurrence sequence τ of Σ be given. Assume that τ is finite. Then, by definition, τ satisfies the progress property (also called the finite delay property) iff its last marking does not activate any pair (t, σ).[10] Moreover, by definition, τ is fair.

[10] I.e., for all t and σ, t is not activated in mode σ.

	In *Impl*:	In *Spec*:
(a)	$(tu_1{:})$ $RemoteCall(p, read, [l])$	$(t_1{:})$ $ReadCall(p, l)$
(b)	$(tu_1{:})$ $RemoteCall(p, write, [l, v])$	$(u_1'{:})$ $WriteCall(p, l, v)$
(c)	$(tu_1{:})$ $RemoteCall(p, read, arglist_0)$	$(t_1{:})$ $ReadCall(p, l_0)$
(d)	$(tu_1{:})$ $RemoteCall(p, write, arglist_0')$	$(u_1'{:})$ $WriteCall(p, l_0, v_0)$
(e)	$(t_B{:})$ $BadCall(p)$	$(v_2{:})$ $MemFail(p)$
(f)	$(t_B{:})$ $BadCall(p)$	$(w_2{:})$ $MemFail(p)$
(g)	$(t_M{:})$ $RPCMemFail(p, read)$	$(v_3{:})$ $MemFail(p)$
(h)	$(t_M{:})$ $RPCMemFail(p, write)$	$(w_3{:})$ $MemFail(p)$
(i)	$(tu_2{:})$ $RemoteReturn(p, read, [l, v])$	$(t_2''{:})$ $ReadReturn(p, l, v)$
(j)	$(tu_2{:})$ $RemoteReturn(p, write, [l])$	$(u_2{:})$ $WriteReturn(p, l)$
(k)	$(tu_2{:})$ $RemoteReturn(p, read, [l, badarg])$	$(v_1{:})$ $BadArg_r(p, l)$
(l)	$(tu_2{:})$ $RemoteReturn(p, write, [l, badarg])$	$(w_1{:})$ $BadArg_w(p, l)$.

Table 1. Identification ρ_{Impl} between *Impl* and *Spec*

Assume that τ is infinite. Then τ satisfies the progress property iff it does not contain an infinite tail such that some pair (t, σ) is permanently activated during this tail by tokens that do not get used otherwise, but (t, σ) does not occur in the tail. Moreover, τ is strongly fair towards some pair (t, σ) iff it has no infinite tail activating (t, σ) infinitely often but not containing (t, σ), and weakly fair towards (t, σ) if it has no infinite tail activating (t, σ) permanently but not containing it. Figure 18 explains the differences between these notions in terms of elementary nets. In Σ_1, the sequence $uuu\ldots$ does not satisfy the progress property (nor is it weakly or strongly fair). In Σ_2, the sequence $uuu\ldots$ satisfies the progress property but is (in both senses) unfair towards t. In Σ_3, the sequence $uu'uu'\ldots$ is weakly but not strongly fair towards t.

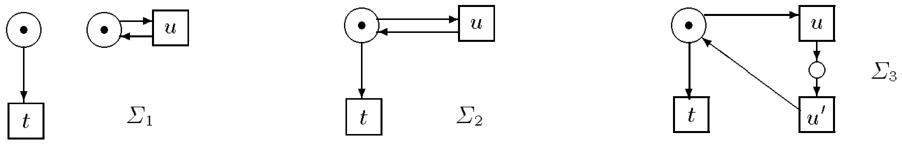


Figure 18. Explanation of progress and fairness

The difference between progress and fairness is related to maximality in partial order semantics. For example, figure 19 shows partial order executions σ_1 and π_2 such that π_1 is non-maximal for Σ_1 (because a t-event can be added) while π_2 is maximal for Σ_2, even though both of them are infinite and have the same transition interleavings.

Claim. If *Retry*, *RPC* and *Spec*3$'$ satisfy the progress property then *Impl* does so, too. More precisely, if τ is an occurrence sequence violating the progress

Figure 19. Two partial order executions π_1 (π_2) of Σ_1 (resp. Σ_2) of figure 18

property with respect to *Impl*, then at least one of the three projections of τ onto the subsystems *Retry*, *RPC* or *Spec*3′ violates progress with respect to the corresponding subsystem. ■5.4

Next, consider the M-net *Impl* in figure 16. We say that an infinite occurrence sequence τ is *read*-complete iff it has no infinite tail which contains an instance of tu_1 with label $\{RemoteCall(p_1, read, [l_1])\}$ but no subsequent instance of tu_2 with label $\{RemoteReturn(p_1, read, [l_1, v])\}$ (with any value v) or instance of t_M with label $\{RPCMemFail(p_1, read)\}$. The notion of τ being *write*-complete is defined analogously. The next property states that when progress is assumed and certain conflict resolutions are assured to be fair, then it is guaranteed that every *read* request will eventually either be served, or an *RPCMemFail* exception be signalled.

Claim. Fairness towards transitions y_1 and tu_2 (figure 16) and the progress property imply that 'A *read* return will eventually be issued'. More precisely, suppose that τ satisfies the progress property and is (weakly) fair towards y_1 and tu_2 (with respect to any binding). Then τ is *read*-complete. ■5.4

Note that fairness towards t_M is not needed. Claim 5.4 gives conditions for

> *'The RPC call can be retried arbitrarily many times because of* RPCFailure *exceptions, but a return from the* Read *or* Write *call must eventually be issued'*

to be true. The next claim states an analogous property for *write* requests.

Claim. Fairness towards z_1', tu_2 and z_2 and the progress property imply that 'A *write* return will eventually be issued'. More precisely, suppose that τ satisfies the progress property and is (weakly) fair towards z_1', tu_2 and z_2 (with respect to any binding). Then τ is *write*-complete. ■5.4

6 A lossy remote procedure call component

In this section we describe an implementation of the *RPC*-module shown in figure 14 by means of a timed lossy module *LRPC*. The latter may exhibit adverse behaviour, in the sense that tokens may get lost in its interior. However, under certain assumptions about its temporal properties, it can be shown that *LRPC* still implements *RPC*. In designing *LRPC*, we make use of the multilabel feature of M-nets.
Section 6.1 describes the enriched temporal model we need. Section 6.2 presents the lossy *RPC* module.

6.1 Time M-nets

We use *time nets* [1, 7, 15, 18] where transitions are given time intervals $[\gamma_1, \gamma_2]$ with numbers γ_1 and γ_2 satisfying $0 \leq \gamma_1 \leq \gamma_2 \leq \infty$. Their semantics can be understood informally as follows. Suppose t is a transition with attached time interval $[\gamma_1, \gamma_2]$. Suppose further that $S_t = {}^\bullet t$ is the set of input places of t. Suppose that a collection of tokens with identity μ on S_t enables t in mode σ. We may imagine that exactly in the moment this enabling starts to exist, a small stopwatch $W(\mu, t)$ appears; this stopwatch refers to μ and to t.[11] $W(\mu, t)$ starts counting time from 0 upwards and disappears as soon as the conditions for its existence cease to be true, i.e. as soon as either t or some other transition take away some token(s) from μ (even if such token(s) are immediately returned to S_t, in which case a new stopwatch appears). For definiteness we assume that time is counted in seconds, and thus γ_1 and γ_2 are either natural numbers or ∞. The rule for the occurrence of t is that

- If $W(\mu, t)$ ceases to exist before it has shown time γ_1 (exclusively) then this must be due to some pair $(t', \sigma') \neq (t, \sigma)$ (note that we may have $t' = t$, but it may not be the identical μ and t);
- $W(\mu, t)$ may not exist after time γ_2 (exclusively). Thus, in particular, if γ_2 is not ∞ then $W(\mu, t)$ must disappear between 0 and time γ_2 (inclusively), be it through occurrence of t in mode σ during $[\gamma_1, \gamma_2]$ or through some other (t', σ') using tokens from μ during $[0, \gamma_2]$.

More informally, this means that 't may not occur through μ before γ_1, and not be enabled by μ after γ_2'. In particular, if $\gamma_1 = \gamma_2$ then t *must* occur at time γ_1 with μ, unless some (other) transition makes $W(\mu, t)$ disappear between time 0 and time γ_1 (inclusively). If $\gamma_1 = \gamma_2 = 0$ then this means that t occurs with μ in the same instant as it is enabled by μ, unless some (other) transition takes μ-tokens away at the same instant. We may also have $\gamma_1 = 0$ and $\gamma_2 = \infty$. This means that no timing constraints are imposed on t. In this way, non-time M-nets are special cases of time M-nets with all intervals being $[0, \infty]$. It may even be the case that $\gamma_1 = \gamma_2 = \infty$, which specifies t as dead. For a more formal definition of time Petri nets, the reader may consult [1, 7, 18, 21, 22].

6.2 Lossy *RPC* module

A specification of the lossy *RPC* interface, *LRPC*, is given in figure 20. For the sake of brevity, we do not give the parameters of action symbols, nor do we specify transition guards; they can (hopefully) be inserted by analogy from previously given ones. The interior interface of *LRPC*, comprising transitions y_1, z_1', q_1, y_2'', z_2 and q_2, is the same as the interior interface of *RPC*. This models

[11] If t is also enabled by a different collection μ' of tokens on S_t then there is a different stopwatch for μ' and t, which may show a different time than $W(\mu, t)$.

'The Lossy RPC component is the same as the RPC component except for the following differences, where δ is a parameter'.

Instead of the *RPCFailure* and *RPCRetry* transitions of RPC, there are now two internal transitions t^2 and t^3 representing the potential loss of tokens. The timing constraints are such that 'loss' transitions t^2 and t^3 can occur at any time if they are enabled. The loss of tokens models

'The RPCFailure *exception is never raised. Instead of raising this exception, the* RemoteCall *procedure never returns'.*

The timing constraint $[0, \delta]$ of t^1 models

'If a call to RemoteCall *raises a* BadCall *exception, then that exception will be raised within δ seconds of the call'.*

The timing constraints $[0, \delta]$ of y_1 and z_1' model

'If a RemoteCall(p, a) *call results in a call of procedure p, then that call of p will occur within δ seconds of the call of* RemoteCall*'.*

Finally, the timing constraint $[0, \delta]$ of t^4 models

'If a RemoteCall(p, a) *call returns other than by raising a* BadCall *exception, then that return will occur within δ seconds of the return from the call to procedure p'.*

Note that the interval $[0, \infty]$ attached to t^2 and t^3 may not be replaced by $[0, \delta]$ because the transitions in conflict with t^2 and t^3 are external, and in a larger net they may not be enabled at the same time as t^2 or t^3; thus, replacing the interval $[0, \infty]$ at t^2 (or t^3) by a smaller one would mean imposing a restriction on the problem which is not implied by its specification.

The task is to implement RPC using a mechanism of passing remote calls through to the $LRPC$ component, and returning normally or raising exceptions:

'The RPC component is implemented with a Lossy RPC component by passing the RemoteCall *call through to the Lossy RPC, passing the return back to the caller, and raising an exception if the corresponding return has not been issued after $2{\cdot}\delta{+}\epsilon$ seconds'.*

We realise this – using multilabels – by the passing-through-module P shown in figure 21. Transition u^3, which must occur exactly at time $2{\cdot}\delta{+}\epsilon{+}1$ after its enabling, unless some other transition (which in this case could only be u^4, since u^2 is not yet enabled at time $2{\cdot}\delta{+}\epsilon$) takes away the enabling token, functions as a timeout for transition u^4. Each occurrence of u^3 takes away one of the enabling tokens of u^4. Then, only u^2 remains enabled, and its time constraint implies that it must occur at some time point strictly after $2{\cdot}\delta{+}\epsilon$ (or not at

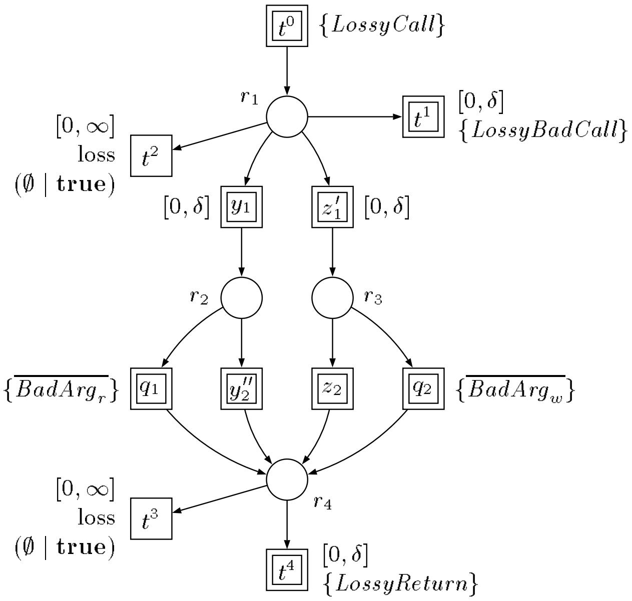


Figure 20. Specification $LRPC$ of Lossy RPC (compare with figure 14)

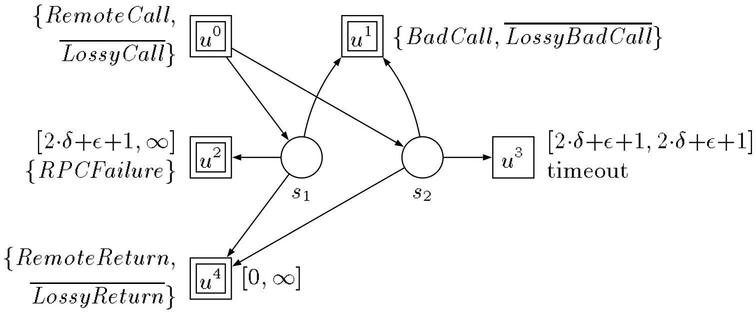


Figure 21. The 'passing through' module P

all). Thus, P has the effect of requiring that, if u^4 misses its chance of occurring between time 0 and time $2{\cdot}\delta{+}\epsilon$, it becomes disabled and u^2 may get executed instead. This models the second part of the above specification.

Observe that u^0 in figure 21 has a label consisting of two action symbols, namely $\{\mathit{RemoteCall}, \overline{\mathit{LossyCall}}\}$. Therefore, u^0 can be synchronised with transition t^0 of LRPC, yielding an external transition having only one action symbol $\mathit{RemoteCall}$ in its label, and thus serving the same purpose as transition tu_1 of RPC. Similarly, transition u^1 can be synchronised with transition t^1 of LRPC and transition u^4 can be synchronised with transition t^4 of LRPC, which realises the first part of the above specification. In all, we have:

$$\begin{aligned} \mathit{RPCImpl} \;\equiv\; (\,(P \parallel \mathit{LRPC})\; & \mathbf{sy}\,\{\mathit{LossyCall}, \mathit{LossyBadCall}, \mathit{LossyReturn}\} \\ & \mathbf{rs}\,\{\mathit{LossyCall}, \mathit{LossyBadCall}, \mathit{LossyReturn}\}\,). \end{aligned}$$

The resulting net is shown in figure 22. Note that the interval of $\mathit{RemoteReturn}$ is the intersection of the two intervals of its constituent transitions.

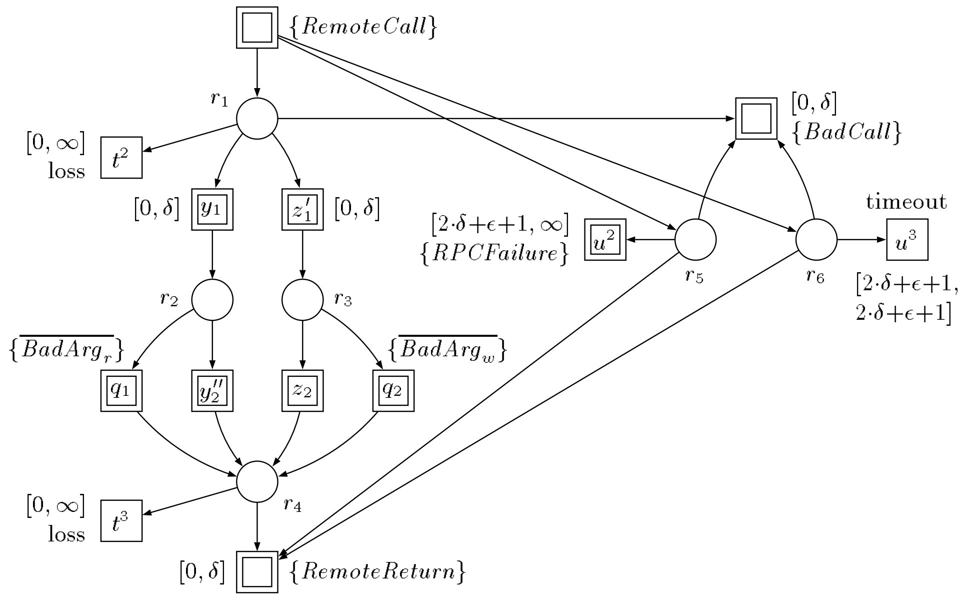


Figure 22. An RPC implementation $\mathit{RPCImpl}$

Finally, let ρ_{RPCImpl} be the obvious inclusion of $\mathit{RPCImpl}$ into RPC.

Claim. If every $\mathit{ReadCall}$ and every $\mathit{WriteCall}$ returns within ϵ seconds, either normally or exceptionally, then $\mathit{RPCImpl} \sqsubseteq_{\rho_{\mathit{RPCImpl}}} \mathit{RPC}$. ■6.2

7 Discussion

7.1 In retrospect

A solution of the RPC-memory specification problem by Broy/Lamport [6] was given in terms of high-level Petri nets, more precisely, M-nets [4]. For the construction of the implementation, composition operations defined on M-nets – in particular, parallel composition, synchronisation and restriction – were used. The construction consisted of a succession of increasingly detailed specifications, which arise by adding the requirements defined in [6] one by one. For the proof that the implementation meets its specification, several simple proof techniques were isolated. In particular, techniques of unfolding-preserving transformations [9, 13], place/transition omission (claims 3.3 and 3.3) and a technique for transition collapsing (claim 3.3) were identified as relevant. The proof consists of repeated applications of these techniques, and only in its last part (showing that $Impl^{(v)}$ implements *Spec*) has a problem-specific argument been used.

What is perhaps nice in this approach is that both the construction of the solution and the proof can be done in small steps which correspond in a meaningful way to aspects and decompositions of the problem. In this way, it is possible to understand each step of the development by itself. I also like to have the specifications and implementations represented as marked nets because their behaviour can readily be visualised and understood.[12]

What is not so nice, perhaps, is that the repetitive parts of the proof do not (yet) seem to be easily automisable. What was also not so nice is that I had to spend quite some time on developing the necessary theory in sections 3.2 and 3.3, even though I did not have to invent an ad-hoc extension of any existing net model, as I had originally feared when first seeing the document [6].

In the development and proof of the solution, there has been no mention of deadlock properties. The reader may have noticed that our notion of collapsing is critical in this respect. Figure 23 shows a pattern of unsafe and safe splittings of transition a. If a is split into an internal action followed by a, as in the left-hand side of the figure, new deadlocks may be introduced in some environments.[13] However, if the split involves a feedback loop (whose pattern is described by transition t_f on the right-hand side of figure 23) then no new deadlocks are introduced. The reader may scan the proof of claim 5.3 below and check that any critical splitting is done under the premise that feedbacks such as t_f are available. Thus, it may be expected that in any environment, our implementation *Impl* is deadlock-free if and only if its specification *Spec* is, although we give no proof of this.

We have also investigated the fairness properties of our solution, emphasising on examining which conditions on its constituent parts are necessary to guarantee some measure of desired service (full service is not guaranteed since there may be

[12] But I agree that such an argument is highly training and experience dependent.
[13] This is analogous to transforming a CCS term $(a+b)$ into the term $((\tau.a)+b)$.

a variety of exceptions). It turns out that with a few fair conflict resolutions, it can be guaranteed that requests will eventually be served. (In practice, instead of implementing fair conflict resolutions involving an exception, one may think of spending an extra effort to avoid exceptions altogether.)

An idea for future work could be to translate all proofs into the language of partial order semantics [2, 10]. This could be beneficial because in our existing proofs, the finger is sometimes pointed at specific tokens m on specific places s, and on a transition t putting m on s and a transition t' removing the very same m from s. While in interleaving semantics this could appear to be a slightly fuzzy notion (particularly since two same elements of a multiset are indistinguishable), it becomes a perfectly clear notion in partial order process (as opposed to trace or pomset [14, 19]) semantics, since m is then represented as a place between two events representing t and t', respectively.

Another idea for future work could be the investigation of congruences regarding the implementation relation $\sqsubseteq_\rho$. For example, it is fairly clear that $\Sigma_1 \sqsubseteq_\rho \Sigma_2$ implies $(\Sigma_1\|\Sigma) \sqsubseteq_\rho (\Sigma_2\|\Sigma)$. It is less clear whether or not similar properties hold for **rs** and **sy**. Such properties are needed, for instance, in order to deduce from claims 5.3 and 6.2 that if *RPC* is replaced by *RPCImpl* in the definition of *Impl*, then claim 5.3 continues to hold.

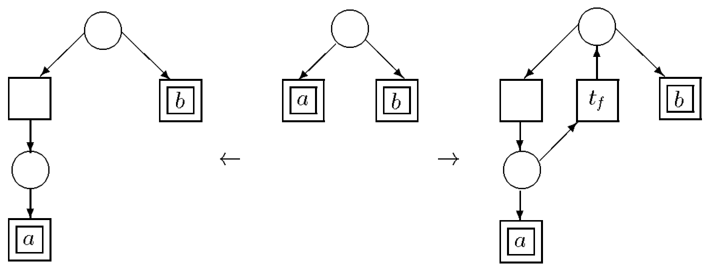


Figure 23. Unsafe (left) and safe (right) splitting of a

7.2 Answers to [6]

> *'Write a formal specification of the Memory component and of the Reliable Memory component'.*

See *Spec*=*Spec*3u (figure 17) and *Spec*3$'$ (figure 13), respectively.

> *'Either prove that a Reliable Memory component is a correct implementation of a Memory component, or explain why it should not be'.*

See claim 4.2; viz., the 'either' case.

'If your specification of the Memory component allows an implementation that does nothing but raise MemFailure *exceptions, explain why this is reasonable'.*

It does. This is due not to the specification, but to the notion of implementation we have defined (i.e. the notion of a relation between two specifications, one of which may then be called an implementation of the other). As discussed above, this notion covers a minimum to be reasonably required of an implementation. A discussion of how it can be strengthened is deferred to section 7.3.

'Write a formal specification of the RPC component'.

See *RPC* in figure 14.

'Write a formal specification of the implementation, and prove that it correctly implements the specification of the Memory component'.

See *Impl* (figure 16) and claim 5.3.

'Write a formal specification of the Lossy RPC component'.

See *LRPC* in figure 20.

'Write a formal specification of this implementation'.

See *RPCImpl* in figure 22.

'Prove that, if every call to a procedure in Procs *returns within ϵ seconds, then the implementation satisfies the specification of the RPC component'.*

See claim 6.2.

7.3 Safety and liveness specifications

As was mentioned in section 3.3, the implementation relation is weak in the sense that, for instance, a net with the empty set of visible traces implements any other net. However, if it is desired to strengthen the notion of implementation to express, say, that some implementation exhibits not just adequate but also desired behaviour, then this can be achieved by using the very same definition in the other direction.

For instance, consider $Spec3'$ which implements $Spec3u$. The latter has additional behaviour which cannot be found in $Spec3'$. Suppose we wish to express that $Spec3'$ covers a certain subset – say, the nonexceptional subset – of the behaviours of $Spec3u$. To express this formally in our example, it is possible to erase the exceptional subset from $Spec3u$, obtaining $Spec3$. To show that $Spec3'$ indeed covers the nonexceptional behaviour of $Spec3u$, it needs to be shown that

*Spec*3 implements *Spec*3′. As this is true, it proves that *Spec*3′ allows all nonexceptional behaviour of *Spec*3*u* (in addition to proving, as was done in the paper, that *Spec*3′ does not allow behaviour disallowed by *Spec*3*u*).

Generalising this observation, if a specification S can be divided into two parts: S_{live}, which specifies behaviour that must necessarily be found in an implementation, and S_{safe}, which specifies behaviour which an implementation must necessarily adhere to, then for some system I to be an implementation satisfying both S_{live} and S_{safe}, it must be shown that

$$S_{live} \sqsubseteq I \sqsubseteq S_{safe}.$$

For instance, with S_{live}=*Spec*3 and S_{safe}=*Spec*3*u*, I=*Spec*3′ satisfies this.

Acknowledgments

I would like to thank Manfred Broy and Leslie Lamport for posing an interesting problem and Arend Rensink and Wolfgang Reisig for making me aware at an early stage (one day after I started to work on the problem) that *Spec*3 does not implement *Spec*2. I am very grateful to Maciej Koutny – who found a major problem in a previous definition of being in sequence – and to Burkhard Graves for careful reading at short notice and spotting several mistakes. Johan Lilius – who suggested a basic pattern leading to the solution described in section 6 – and Peter Starke helped me out in the jungle of time(d) Petri nets. Thanks are also due to Hanna Klaudel, Javier Esparza, Hans Fleischhack and two anonymous reviewers for comments on a draft of this paper. Carola Pohl created most of the figures using the 'graphs.sty' package by Frank Drewes.

References

1. B. Berthomieu and M. Diaz: Modeling and Verification of Time Dependent Systems Using Time Petri Nets. IEEE Transactions on Software Engineering Vol. 17/3, 259–273 (1991).
2. E. Best and R. Devillers: Sequential and Concurrent Behaviour in Petri Net Theory. Theoretical Computer Science Vol. 55/1, 87–136 (1988).
3. E. Best, R. Devillers and J.G. Hall: The Petri Box Calculus: a New Causal Algebra with Multilabel Communication. Advances in Petri Nets 1992 (ed. G.Rozenberg), Springer-Verlag, LNCS Vol. 609, 21–69 (1992).
4. E. Best, H. Fleischhack, W. Frączak, R.P. Hopkins, H. Klaudel and E. Pelz: A Class of Composable High Level Petri Nets. Proceedings of Petri Nets'95, Torino, G. De Michelis, M. Diaz (eds), Springer-Verlag, LNCS Vol.935, 103–120 (1995).
5. E. Best and Th. Thielke: Coloured Nets with Curry. Petri Net Newsletters No. 50, 27–44 (1996).
6. M. Broy and L. Lamport: Specification Problem. Basic document for the Dagstuhl Workshop (December 1993/September 1994).
7. A. Cerone and A. Maggiolo-Schettini: Time-based Expressivity of Timed Petri Nets. TR 12/95, Goethe-Universität, Frankfurt am Main (1995).
8. H.J. Genrich: Predicate-transition Nets. Advances in Petri Nets 1986, Springer-Verlag, LNCS Vol.254, 207–247 (1987).

9. H.J. Genrich: Equivalence Preserving Transformations of Predicate-transition Nets, Arbeitspapiere der GMD (1987).
10. U. Goltz and W. Reisig: The Non-sequential Behaviour of Petri Nets. Information and Control Vol. 57/2-3, 125–147 (1983).
11. K. Jensen: *Colored Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Volume 1. EATCS Monographs on Theoretical Computer Science, 1992.
12. H. Klaudel: Modèles algébriques, basés sur les réseaux de Petri, pour la sémantique des langages de programmation concurrents. PhD Thesis, Université Paris-Sud, Orsay, 1995.
13. H. Klaudel, R.-C. Riemann and A. Gronewold: Consistent Equivalence Notion for a Class of High Level Petri Nets. Rapport de Recherche 1054, LRI, Université Paris Sud (June 1996).
14. A. Mazurkiewicz: Concurrent Program Schemes and Their Interpretation. University of Århus, Computer Science Department, DAIMI PB-78 (1977).
15. P.M. Merlin and D.J. Farber: Recoverability of Communication Protocols - Implication of a Theoretical Study. IEEE Transactions on Software Communications 24, 1036–1043 (1976).
16. A.J.R.G. Milner: *Communication and Concurrency.* Prentice Hall (1989).
17. E.R. Olderog: *Nets, Terms and Formulas.* Cambridge Tracts in Theoretical Computer Science Vol. 23 (1991).
18. L. Popova: On Time Petri Nets. Journal of Information Processing and Cybernetics (formerly EIK), Vol. 27/4, 227–244 (1991).
19. V.R. Pratt: The Pomset Model of Parallel Processes: Unifying the Temporal and the Spatial. Springer-Verlag, LNCS Vol. 197: *Seminar on Concurrency*, S.D. Brookes, A.W. Roscoe, G. Winskel (eds), 180–196 (1985).
20. W. Reisig: *Petri Nets. An Introduction.* EATCS Monographs on Theoretical Computer Science Vol. 4 (1985).
21. P.H. Starke: *Analyse von Petri-Netz-Modellen.* Verlag Teubner, Stuttgart (1990).
22. P.H. Starke: A Memo on Time Constraints in Petri Nets. Informatik-Bericht Nr. 46, Humboldt-Universität zu Berlin (August 1995).

A Proofs

Proof of claim 3.2

Let T and T' be in sequence in Σ. By definition, either $\forall t{\in}T{:}\, t^\bullet{=}\{s_{TT'}\}$ or $\forall t'{\in}T'{:}\, {}^\bullet t'{=}\{s_{TT'}\}$. Assume that ${}^\bullet t'{=}\{s_{TT'}\}$ for all $t'{\in}T'$. Let

$$\tau \;=\; \ldots\; M_{i-1}\; \underbrace{(t,\sigma)}_{(1),i}\; M_i\; \underbrace{\ldots\;\;\ldots}_{(2)}\; M_{j-1}\; \underbrace{(t',\sigma')}_{(3),j}\; M_j\; \ldots$$

be such that $t{\in}T$, by (1), puts some token m on $s_{TT'}$ which remains there during (2) until it is removed by $t' \in T'$, by (3). We show that by a sequence of $\equiv_0$-permutations, (t',σ') can be moved leftward into a position immediately following (t,σ). Note that in (2) there could be other instances of t or t' (or any other transition of T or T', for that matter). Hence to be precise we consider indices into τ. Let i and j be the initial indices of (t,σ) and (t',σ'), respectively.

If $i{=}j{-}1$ then there is nothing to prove. So assume $i{<}j{-}1$ and consider the index $j{-}1$. Let (x,ξ) be the transition and binding at index $j{-}1$.

Assume first that $x{\notin}T'$. An $\equiv_0$-permutation of the pairs at indices $j{-}1$ and j could be hindered only by some output place of x, say s, which is also an input place of t', such that x puts a token on this place needed by t' for its occurrence. By the assumption ${}^\bullet t'{=}\{s_{TT'}\}$, $s{=}s_{TT'}$, and since t' needs only the token m for its occurrence, the token put there by x must be a different one. Hence the marking M_{j-2} enables (x,ξ) and (t',σ') concurrently, and thus they can be $\equiv_0$-permuted.

Assume next that $x{\in}T'$ and hence, ${}^\bullet x{=}\{s_{TT'}\}$ by the above assumption. Because the token m on $s_{TT'}$ removed by t' at index j remains on $s_{TT'}$ all through (2), the token taken from $s_{TT'}$ by x is a different one (even if x is t'). From the fact that x and t' have no other common input places apart from $s_{TT'}$, it follows that the marking M_{j-2} just before (x,ξ) enables (x,ξ) and (t',σ') concurrently, and (x,ξ) and (t',σ') can thus be $\equiv_0$-permuted.

This procedure can be repeated. Moreover, moving t' leftward may temporarily destroy already established neighbourships, say of u and u', but a permutation of t' with u' is always directly followed by another permutation of t' with u, so that neighbourship between u and u' is instantly re-established. Moreover, as a consequence of the fact that t' takes away no more than one token from $s_{TT'}$, once t' has been moved to the right of its corresponding t, there can be no other transition that would lead to any further moving of t'.

Finally, suppose that $s_{TT'}^\bullet{=}T' \wedge \forall t{\in}T{:}\, t^\bullet{=}\{s_{TT'}\}$, rather than $\forall t'{\in}T'{:}\, {}^\bullet t'{=}\{s_{TT'}\}$ holds true. Then a symmetric argument shows that (t,σ) can be moved to the right until it is a neighbour of (t',σ'). ■3.2

Proof of claim 3.3

Let τ_1 be a δ-complete sequence of Σ_1. We construct a sequence τ_2 of Σ_2 such that τ_1^δ and τ_2 are $\delta\rho$-related. We do this inductively for finite sequences and appeal to a standard generalisation for infinite sequences. Assume

$$\tau_1^\delta = M^0(t^1,\sigma^1)M^1 \ldots M^{n-1}(t^n,\sigma^n)M^n.$$

Note that markings M of Σ_1 such that $M(s){=}\emptyset$ and markings K of Σ_2 correspond uniquely to each other.

The construction of τ_2 begins by letting K^0 be the marking corresponding to M^0. Then we distinguish two cases:

Case 1: $t^1{\neq}t$. Let $x^1{=}t^1$ and $\xi_1{=}\sigma_1$ and K^1 the marking corresponding to M^1 and continue the construction with the tail $M^1\ldots$ of τ_1^δ. (Note that by δ-completeness of τ_1^δ, $t^1{\notin}T'$.)

Case 2: $t^1{=}t$. By δ-respectfulness of τ_1^δ, t^2 exists and satisfies $t^2{\in}T'$. Let $x^1{=}t^1t^2$ (i.e. the transition in Σ_2 which has been combined from t^1 and t^2) and let ξ_1 be $\sigma_1{\cup}\sigma_2$. Let K^1 be the marking corresponding to M^2 and continue the construction with the tail $M^2\ldots$ of τ_1^δ.

Continuing in this way (either, in case 1, translating transitions of τ_1^δ one-to-one into transitions of τ_2, or, in case 2, combining two successive transitions of τ_1^δ into a single transition of τ_2), one eventually gets $\tau_2{=}K^0(x_1,\xi_1)K^1\ldots$ which is $\delta\rho$-related to τ_1^δ. ■3.3

Proof of claim 4.2

*Spec*0$\sqsupseteq$*Spec*1 follows with claim 3.3. The proof of *Spec*1$\sqsupseteq$*Spec*2 is a repeated application of claim 3.3, first applying it to $\{t_2'\}$ and $\{t_2''\}$ and then to $\{u_1'\}$ and $\{u_1''\}$ (or, what comes to the same, the other way round). *Spec*2 $\sqsubseteq$ *Spec*3 is a consequence of claim 3.3. *Spec***i** $\sqsubseteq$ *Spec***i***u* (for all **i**$=0,1,2,3$) also follow from claim 3.3. The proofs of the claims of the middle and lower lines of the claim mirror those of the upper line. ■4.2

Proof of claim 5.3

The task is to prove that every occurrence sequence of *Impl* has a corresponding sequence in *Spec*. First, we explain informally how *Impl* and *Spec* correspond to each other. Their interface transitions are related by ρ_{Impl} as defined in table 1. Moreover, the internal transitions of *Spec* are 'somehow' (in a sense to be made precise) simulated by transitions of *Impl*. For example, transition t_2' of *Spec* corresponds to transitions y_1, y_2' and y_2'' of *Impl*. Similarly, u_1'' corresponds to z_1', z_1'' and z_2. In addition, *Impl* contains a feedback loop through transitions t_F and t_R, which can be entered after a *read*, after a *write* or after *BadArg* exceptions q_1 or q_2. This loop can 'somehow' be retrieved in *Spec* by transition $\widetilde{u_1''}$, but in the *read* part of *Spec* there is no such loop. Thus, we need to show its 'harmlessness'. Furthermore, *Impl* contains other intermediate behaviour which needs to be related to behaviour of *Spec*. It is here that the theory of sections 3.2 and 3.3 helps. We will use claims 3.3–3.3 in order to construct a series of marked M-nets $\Sigma^0,\ldots,\Sigma^m$ such that $Impl{=}\Sigma^0$ and $\Sigma^m{=}Spec$, and Σ^{j-1} implements Σ^j for $0{<}j{\le}m$.

To make the correspondence between the *RemoteCall* transition tu_1 of figure 16 with its two parameters *read* and *write* and the two transitions *ReadCall* (t_1) and *WriteCall* (u_1') of figure 17 transparent, we use unfolding preserving transformations, all of which are incarnations of the principle explained in section 2.2 (cf. figure 3). Such transformations involve splitting of transitions and/or places such that the resulting net has the same unfolding as the original net. In our case, they are driven by suitable partitionings of the type sets of places.

After suitable transformations of this kind, the M-net *Impl*$'$ depicted in figure 24 (with inscriptions as in table 2)[14] is obtained from *Impl* by splitting place s_1 into s_1^{rb}, s_1^r, s_1^w and s_1^{wb} and place s_6 into s_6^r, s_6^{rb}, s_6^{wb} and s_6^w, and correspondingly

[14] Actually, table 2 shows only part of the *read* part of the figure. The *write* part is, *mutatis mutandis*, analogous. We use the abbreviation $\{[l]\}{=}\{[l]\mid l{\in}(Loc{\cup}NonLoc)\}$, and similarly $\{[l,v]\}{=}\{[l,v]\mid l{\in}(Loc{\cup}NonLoc)\wedge v{\in}\,Val\}$.

splitting transition tu_1 into t_1^b, t_1, u_1' and $u_1'^b$ and tu_2 into t_2'', t_2^b, u_2^b and u_2. *Impl'* has the same unfolding as *Impl*, and hence *Impl* implements *Impl'*. The annotations (a)–(l) of figure 24 refer back to the lines of table 1.

$$
\begin{array}{ll}
\iota(t_1^b) & = (\,\{RemoteCall(p, read, arglist_0)\} \mid b{=}(p, read, arglist_0)\,) \\
\iota(t_1) & = (\,\{RemoteCall(p, read, [l])\} \mid b{=}(p, read, [l])\,) \\
\iota(x^1) & = (\,\{BadCall(p)\} \mid a{=}(p, read, arglist_0)\,) \;\; (BADCALL \text{ is true, cf. table 1}) \\
\iota(s_1^{rb}) & = Pcss \times \{read\} \times (Args \backslash \{[l]\}) \\
\iota(s_1^r) & = Pcss \times \{read\} \times \{[l]\} \\
\iota(s_0^r) & = Pcss \times \{read\} \times Args \\
\iota(y_2') & = (\,\emptyset \mid a{=}(p,l) \wedge c{=}(l,v) \wedge b{=}(p,l,v) \wedge d{=}(l,v)\,) \\
\iota(y_2'') & = (\,\emptyset \mid a{=}(p,l) \wedge c{=}(p,l,v) \wedge b{=}(p, read, l, v)\,) \\
\iota(\widetilde{q}_1) & = (\,\emptyset \mid a{=}(p,l){=}c \wedge b{=}(p, read, l, badarg) \wedge l{\in}NonLoc\,) \\
\iota(t_M^r) & = (\,\{RPCMemFail(p, read)\} \mid a{=}(p, r, arglist)\,) \\
\iota(s_6^r) & = Pcss \times \{read\} \times Loc \times Val \\
\iota(s_6^{rb}) & = Pcss \times \{read\} \times NonLoc \times \{badarg\} \\
\iota(t_2'') & = (\,\{RemoteReturn(p, read, [l, v])\} \mid a{=}(p, read, l, v) \wedge v{\in}Val\,) \\
\iota(t_2^b) & = (\,\{RemoteReturn(p, read, [l, badarg])\} \mid a{=}(p, read, l, badarg)\,).
\end{array}
$$

Table 2. Some inscriptions of figure 24 (*Impl'*)

Next, consider the M-net obtained from *Impl'* by omitting places s_7 and s_8. By claim 3.3, this net is implemented by *Impl'* (and hence also by *Impl*). After removal of s_7 and s_8, $\{y_2'\}$ and $\{y_2'', \widetilde{q}_1\}$ are in sequence (as they are sequentialised by place $s_{\{y_2'\}\{y_2''\}}{=}s_4$) and $\{z_1'\}$ and $\{z_1'', \widetilde{q}_2\}$ are in sequence (as they are sequentialised by place s_3). Moreover, s_6^{rb} sequentialises $\{\widetilde{q}_1\}$ and $\{t_2^b\}$ and s_6^{wb} sequentialises $\{\widetilde{q}_2\}$ and $\{u_2^b\}$.

Applying claim 3.3 four times shows that y_2', s_4 and y_2'' and $\widetilde{q}_1$, s_6^{rb} and t_2^b on the one hand, and z_1', s_3 and z_1'' and $\widetilde{q}_2$, s_6^{wb} and u_2^b on the other hand, can be replaced by single transitions, say y_2, t_2^b, z_1 and u_2^b, respectively. Also, the singleton factor $\{read\}$ can be omitted at place s_6^r (this is justified by the fact that unfolding is stable with respect to isomorphic type changes, see section 2.2). The result is *Impl''* as shown in figure 25. Claim 3.3 shows that *Impl'* implements *Impl''* while (as will be seen later) *Impl''* still implements *Spec*.

Next, consider *Impl''*. Observe that $\{z_2\}$ and $\{u_2, t_F^w\}$ are in sequence by s_6^w. According to claim 3.3, they can be replaced by two transitions. Moreover, after suitably splitting places s_0^r and s_0^w, the 'null loops' between those places and s_1^{rb}, s_1^r and s_1^w, s_1^{wb} can be collapsed as shown at the end of section 3.3 (cf. figure 7). This yields *Impl'''* as shown in figure 26.[15] It is still true that *Impl*

[15] In going from *Impl''* to *Impl'''*, we have renamed y_2 into t_2', z_1 into u_1'' and $\widetilde{z_1''}$ into $\widetilde{u_1''}$. This has been done in order to bring *Impl'''* closer to *Spec*. Moreover, transition t_F^w in *Impl'''* is really the collapsed counterpart of $[z_2, t_F^w]$ of *Impl''*. Similarly, u_2 in *Impl'''* is the counterpart of $[z_2, u_2]$ of *Impl''*.

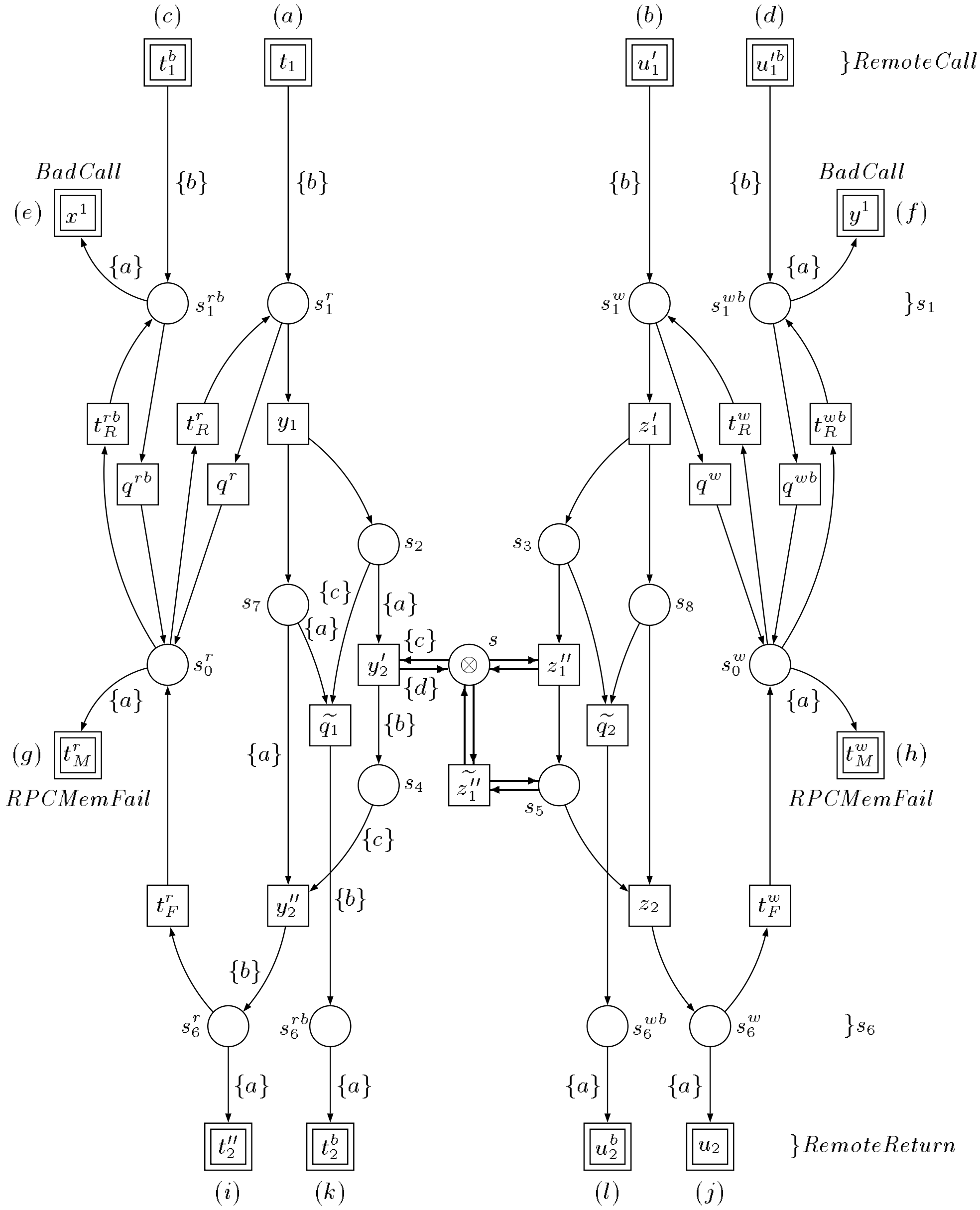
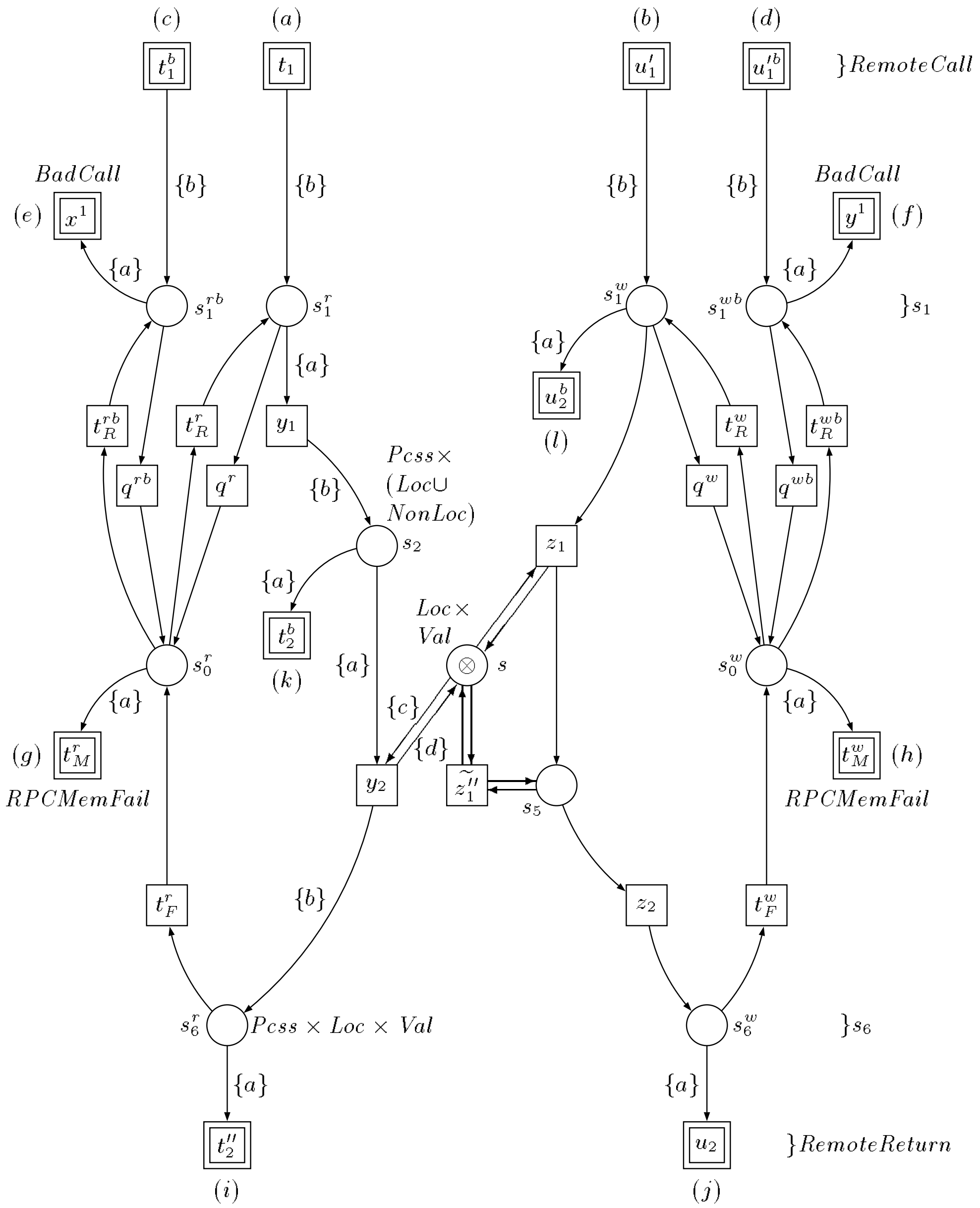


Figure 24. Implementation *Impl′*



$\iota(s_1^w) \ = Pcss \times \{write\} \times \{[l, v]\}$
$\iota(u_2^b) \ = (\ \{RemoteReturn(p, write, [l, badarg])\} \mid a{=}(p, write, [l, v]) \wedge l{\in}NonLoc\ )$
$\iota(y_2) \ = (\ \emptyset \mid a{=}(p, l) \wedge c{=}(l, v) \wedge b{=}(p, l, v) \wedge d{=}(l, v)\ )$
$\iota(t_2^b) \ = (\ \{RemoteReturn(p, read, [l, badarg])\} \mid a{=}(p, l) \wedge l{\in}NonLoc\ )$
$\iota(s_1^r) \ = Pcss \times \{read\} \times \{[l]\}$
$\iota(y_1) \ = (\ \emptyset \mid a{=}(p, read, [l]) \wedge b{=}(p, l)\ ).$

Figure 25. Implementation $Impl''$ and some of its inscriptions

implements $Impl'''$.

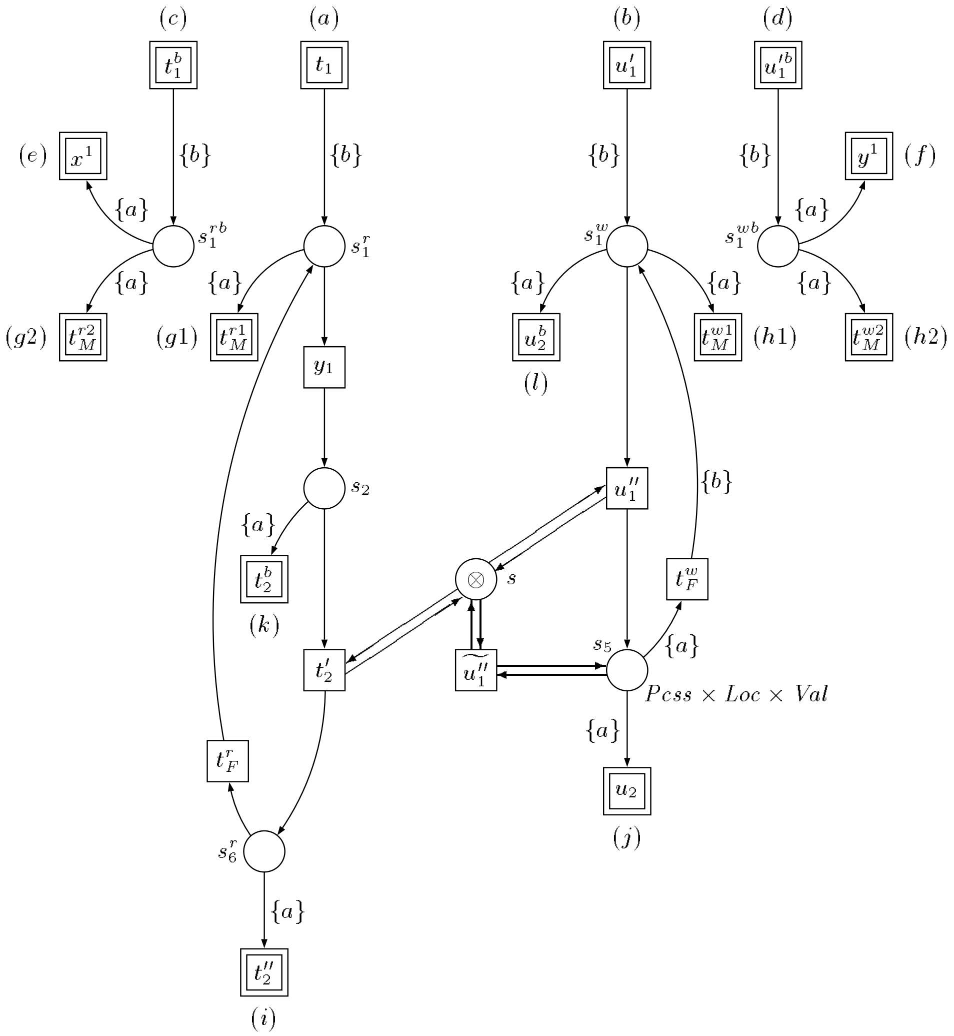
In the next step, consider $Impl'''$. The sets $\{y_1\}$ and $\{t_2^b, t_2'\}$ are in sequence by s_2; so they can be replaced by two transitions. Moreover, the types of the places $s_1^{rb}, \ldots, s_1^{wb}$ can be modified by identifying all tokens $(p, \mathit{read}, \mathit{arglist}_0)$ on s_1^{rb} with (p, l_0) and identifying all tokens $(p, \mathit{write}, \mathit{arglist}_0')$ on s_1^{rb} with (p, l_0, v_0), as in table 1. This is justified by the fact that the parameters $\mathit{arglist}_0$ and $\mathit{arglist}_0'$ are 'forgotten' in the labels of x^1, y^1 (*BadCall*) and t_M^{r2}, t_M^{w2} (*RPCMemFail*). Moreover, the factors $\{\mathit{read}\}$ and $\{\mathit{write}\}$ can be neglected in the types of places s_1^{rb}, s_1^r, s_1^w and s_1^{wb}, which is justified by the fact, pronounced at the end of section 2.2, that unfolding is stable with respect to isomorphic type changes. Eventually, this yields the M-net $Impl^{(iv)}$ shown in figure 27. By claim 3.3 (and other claims stated in sections 2 and 3), *Impl* implements $Impl^{(iv)}$.

$Impl^{(iv)}$ still contains internal loops (t_2', t_F^r and u_1'', t_F^w) that do not directly correspond to a part of *Spec*. We apply partial collapsing, as provided by claim 3.3, in order to transform $Impl^{(iv)}$ into yet another M-net which is even closer to *Spec*. To this end we exploit the fact that, in $Impl^{(iv)}$, s_1^r sequentialises $\{t_1, t_F^r\}$ and $\{t_2', t_M^{r1}, t_2^b\}$, and s_1^w sequentialises $\{u_1', t_F^w\}$ and $\{u_1'', t_M^{w1}, u_2^b\}$. Of the pairs that can be collapsed in each sequence, we collapse $[t_F^r, t_2']$, $[t_F^r, t_M^{r1}]$ and $[t_F^r, t_2^b]$ in the first sequence and $[t_F^w, u_1'']$, $[t_F^w, t_M^{w1}]$ and $[t_F^w, u_2^b]$ in the second sequence. Note that $[t_F^r, t_2^b]$ and $[t_F^w, u_2^b]$ disappear because their guards are **false**. We also recombine places s_1^{rb} and s_1^r (in $Impl^{(iv)}$) into place s_1^r (in $Impl^{(v)}$) and places s_1^{wb} and s_1^w (in $Impl^{(iv)}$) into place s_1^w (in $Impl^{(v)}$). $Impl^{(v)}$ is shown in figure 28. Transition x corresponds to $[t_F^r, t_2']$ and transition y corresponds to $[t_F^w, u_1'']$, while transition t_M^r corresponds to $[t_F^r, t_M^{r1}]$ and transition t_M^w corresponds to $[t_F^w, t_M^{w1}]$.

The relationship between $Impl^{(v)}$ and *Spec* is indicated by the $\hat{=}$ in figure 28. We are left with needing to prove that $Impl^{(v)}$ implements *Spec*, i.e. that every occurrence sequence of $Impl^{(v)}$ has a matching (i.e. equivalent on the visible interface) sequence in *Spec*. A difference between $Impl^{(v)}$ and *Spec* is that $Impl^{(v)}$ contains internal transitions x and y. To cope with this, we examine the origins of transitions x and y in order to understand their role in $Impl^{(v)}$.

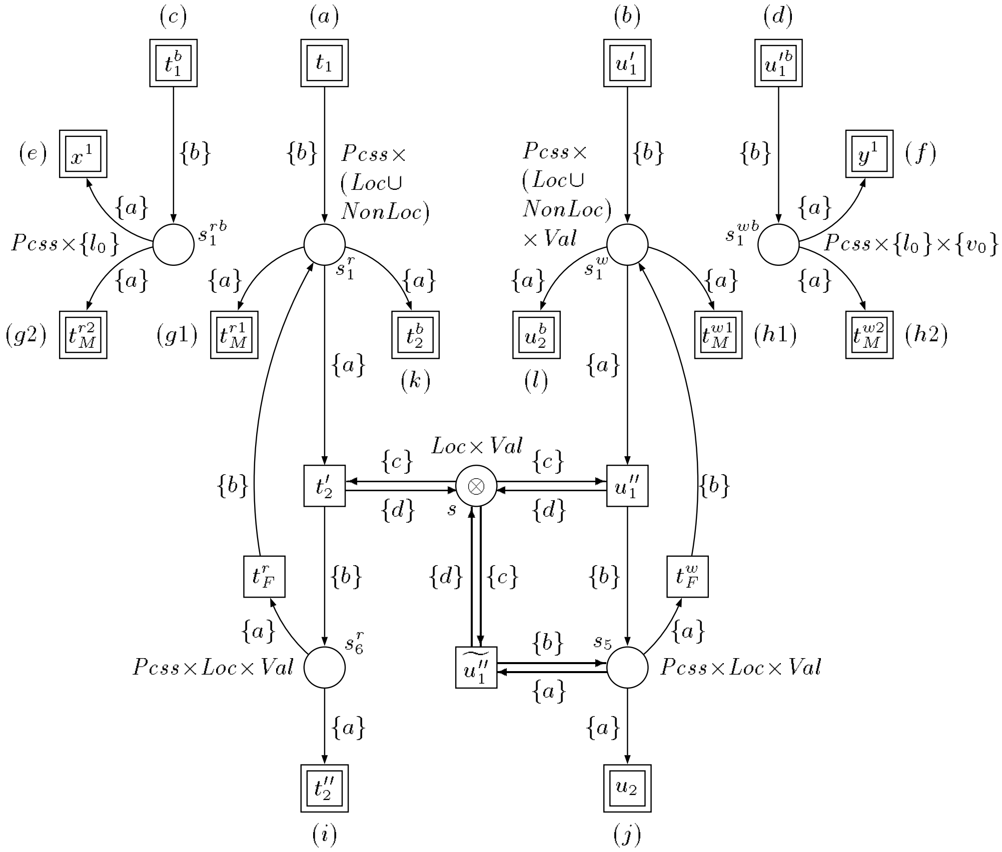
Consider first y. By the rules of synchronisation and collapsing, and considering its origins, y turns out to be an exact copy of the 'repeated actual write' transition $\widetilde{u_1''}$. Thus, when transforming a given occurrence sequence τ of $Impl^{(v)}$ into one of *Spec*, we may simply replace every instance of y by an instance of $\widetilde{u_1''}$.

Consider next x. It takes a token $m_1{=}(p_1, l_1, w)$ from s_6^r and a token $m_2{=}(l_2, v_2)$ from s and returns a token $m_3{=}(p_3, l_3, v_3)$ to s_6^r and a token $m_3{=}(l_4, v_4)$ to s. Token m_1 is a request of process p_1 to repeat the reading of location l_1 and token m_2 describes the current value of location l_2. The value w represents the value last read by this request (which may be different from v_2, the actual value). The inscription of x shows that it identifies $p_1{=}p_3$ and $l_1{=}l_2{=}l_3{=}l_4$, as well as values $v_2{=}v_3{=}v_4$. In particular, the token m_1 taken from place s_6^r may be different from token m_3 put back on that place. (Notice the difference to y where the



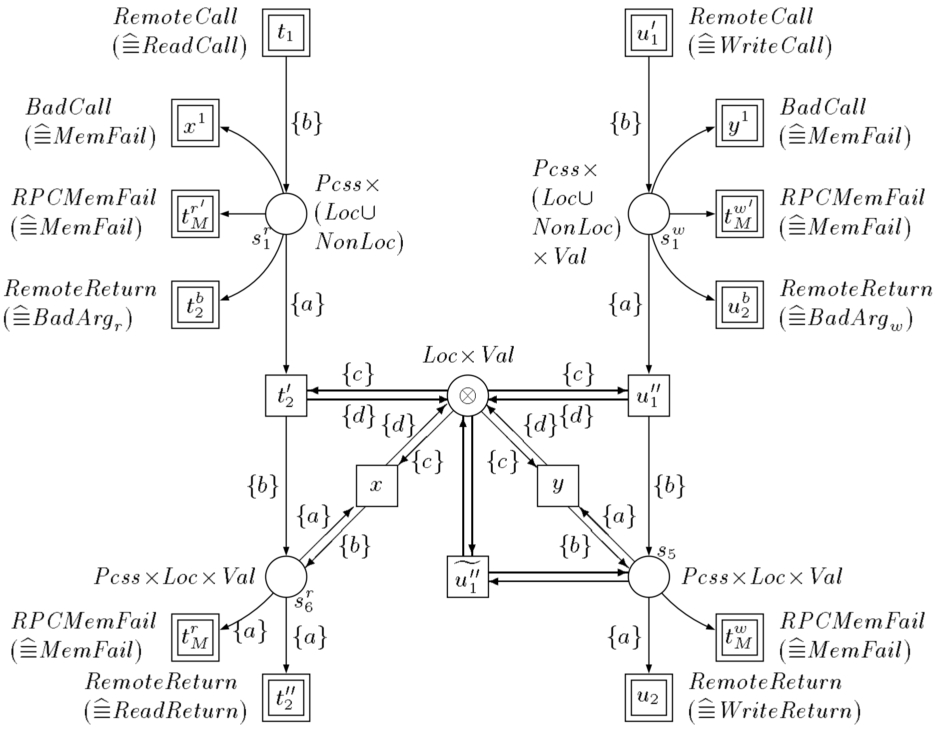
$$
\begin{aligned}
\iota(s_1^{rb}) &= Pcss \times \{read\} \times (Args \backslash \{[l]\}) \\
\iota(x^1) &= (\,\{BadCall(p)\} \mid a{=}(p, read, arglist_0)\,) \\
\iota(t_M^{r2}) &= (\,\{RPCMemFail(p, read)\} \mid a{=}(p, read, arglist_0)\,) \\
\iota(t_M^{r1}) &= (\,\{RPCMemFail(p, read)\} \mid a{=}(p, read, l)\,) \\
\iota(s_1^r) &= Pcss \times \{read\} \times \{[l]\} \\
\iota(s_1^w) &= Pcss \times \{write\} \times \{[l, v]\} \\
\iota(s_1^{wb}) &= Pcss \times \{write\} \times (Args \backslash \{[l, v]\}) \\
\iota(s_5) &= Pcss \times Loc \times Val \\
\iota(t_F^w) &= (\,\emptyset \mid a{=}(p, l, v) \wedge b{=}(p, write, [l, v])\,) \\
\iota(u_2) &= (\,\{RemoteReturn(p, write, [l])\} \mid a{=}(p, l, v)\,).
\end{aligned}
$$

Figure 26. Implementation *Impl'''* and some of its inscriptions



$\iota(t_2') = (\,\emptyset \mid a{=}(p,l) \wedge l{\in}Loc \wedge c{=}(l,v) \wedge b{=}(p,l,v) \wedge d{=}(l,v)\,)$
$\iota(t_2^b) = (\,\{RemoteReturn(p, read, [l, badarg])\} \mid a{=}(p,l) \wedge l{\in}NonLoc\,).$

Figure 27. Implementation $Impl^{(iv)}$ and some of its inscriptions



$$
\begin{aligned}
\iota(t_2') &= (\,\emptyset \mid a=(p,l) \wedge c=(l,v) \wedge b=(p,l,v) \wedge d=(l,v)\,)\\
\iota(u_1'') &= (\,\emptyset \mid a=(p,l,v) \wedge c=(l,w) \wedge b=(p,l,v) \wedge d=(l,v)\,)\\
\iota(\widetilde{u_1''}) &= (\,\emptyset \mid a=(p,l,v) \wedge c=(l,w) \wedge b=(p,l,v) \wedge d=(l,v)\,)\\
\iota(x) &= (\,\emptyset \mid a=(p,l,w) \wedge c=(l,v) \wedge b=(p,l,v) \wedge d=(l,v)\,)\\
\iota(y) &= (\,\emptyset \mid a=(p,l,v) \wedge c=(l,w) \wedge b=(p,l,v) \wedge d=(l,v)\,)\\
\iota(t_M^r) &= (\,\{RPCMemFail(p, read)\} \mid a=(p,l,v)\,).
\end{aligned}
$$

Figure 28. Implementation $Impl^{(v)}$ and some of its inscriptions

token taken from s, rather than the token taken from s_5, may be different from the one put back there.)

Now let τ be an occurrence sequence of $Impl^{(v)}$. We will gradually transform τ into a matching occurrence sequence of *Spec*. By the previous considerations, we may assume that τ contains no instances of y. Furthermore, instances of t^r_M and t^w_M of $Impl^{(v)}$ may be identified with instances of v_3 and w_3, respectively, of *Spec*, by (g) and (h). A similar remark is true for instances of x^1, $t^{r'}_M$, t^b_2, y^1, $t^{w'}_M$ and u^b_2 of $Impl^{(iv)}$. If τ contains no instances of x then there is nothing more to prove. So, assume now that τ contains at least one instance of x, say at index i, and with input tokens $m_1{=}(p_1, l_1, w)$ and $m_2{=}(l_1, v_1)$ and output tokens $m_3{=}(p_1, l_1, v_1)$ and $m_4{=}(l_1, v_1)$.

Suppose first that after this instance, τ contains some instance, say at index i, of either t^r_M or t''_2 with parameters (p_1, l_1, w') (some arbitrary value w' which may be different from w). Let j be a minimal index of such an occurrence after index i. Because place s^r_6 is empty in the initial marking, some token with parameters p_1 and l_1 has been put there by some occurrence of transition t'_2. Let k be a maximal index of such an occurrence before index i. Then the section from k to j in τ can be interpreted in the following way: k is the initial production of a token $(p_1, l_1, .)$ on place s^r_6, all x-instances between k (exclusively) and j (of which there is at least one) are repeated reads of the same location by the same process, and j is the eventual consumption of token $(p_1, l_1, .)$ from place s^r_6. In particular, between k and j, no transition instance (other than the x instances and the instance of t^r_M or t''_2 at j) ever needs a token $(p_1, l_1, .)$ for its occurrence. It follows that when between k (inclusively) and j all instances of t'_2 and x are erased, except for the last x instance, which is replaced by an instance of t'_2, then the resulting sequence does not lose its property of being an occurrence sequence.

Suppose next (and last) that after the assumed instance of x at index i, there is no further instance of either t^r_M or t''_2 with parameters p_1 and l_1. Then by the same reasoning, it follows that in the entire tail of τ, tokens with parameters $(p_1, l_1, .)$ on s^r_6 are used at most for instances of x. It follows that all such instances can be erased from τ without destroying its property of being an occurrence sequence.

This shows, then, that $Impl^{(v)}$ implements *Spec* (with respect to empty δ and the obvious ρ). In sum, the proof has exhibited a sequence of M-nets, from *Impl*, *Impl'* to $Impl^{(v)}$, and finally *Spec*, such that every element of the sequence implements its successor (if there is one), proving, in all, that *Impl* implements *Spec* as required. ■5.3

Note that the last step of the proof cannot be applied to transition $\widetilde{u''_1}$ in the same way as to x. The main reason is that x modifies only place s^r_6, which can be kept under control as shown above, but $\widetilde{u''_1}$ modifies place s upon which other transitions such as t'_2 depend, and hence the argument which was valid for x is invalid for $\widetilde{u''_1}$. In particular, it cannot be proved (and is indeed untrue) that *Impl* implements *Spec2u*, i.e. unreliable memory without repeated writes.

Proof of claim 5.4

This is a general property of nets which are composed from others by synchronisation and restriction. Suppose that in the composed net some transition t does not progress. If t is a member of one of the constituent nets then this contradicts the progress property of the constituent net. If t is a synchronisation, say of two transitions of two constituent nets, then this contradicts the progress property for both of them. ■5.4

Proof of claim 5.4

Suppose that τ is not *read*-complete. Then there is some last instance of tu_1 with label $\{RemoteCall(p_1, read, [l_1])\}$ in τ, such that τ contains no subsequent occurrence of tu_2 with the same parameters p_1 and $[l_1]$ (and some arbitrary value v_1). By the structure of the net, this implies that in every marking of the tail of τ determined by this last occurrence of tu_1, there is either a token $(p_1, read, [l_1])$ on s_1; or tokens (p_1, l_1) and (p_1, l_1) on s_7 and s_2, respectively; or tokens (p_1, l_1) and (p_1, l_1, v_1) on s_7 and s_4, respectively; or a token $(p_1, read, l_1, v_1)$ on s_6; or a token $(p_1, read, [l_1])$ on s_0. In the first case, this token activates y_1 or q (under the induced binding), and in the other cases, y_2', y_2'', both t_F and tu_2, and t_R are activated, respectively. Progress and the fact that the tail of τ in question does not contain tu_2 for this particular binding imply that the tail of τ contains an infinity of instances of the form $qt_R \ldots$ or $y_1 y_2' y_2'' t_F t_R \ldots$, all with the same bindings. This contradicts fairness towards y_1 or fairness towards tu_2. ■5.4

Proof of claim 6.2

It has to be shown that the observable part of a behaviour τ of *RPCImpl* can – under the premise that tokens may not rest on places r_2 and r_3 longer than time ϵ – be retrieved from some behaviour of *RPC*. This can be seen as follows. Initially, the *RemoteCall* transition of *RPCImpl* puts three similar tokens on r_1, r_5 and r_6, say, respectively, m, n and n'. Thereafter, there are three alternatives for m, n and n'.

Firstly, m might get passed through to r_4, either via r_2 or via r_3. When this happens, m appears on r_4 within time $\delta+\epsilon$, since it may stay on r_1 not longer than δ and on r_2 or r_3 not longer than ϵ. In that case, n and n' are still present (at least for time δ) on r_5 and r_6, and thus the *RemoteReturn* transition is enabled for a time span of δ and may occur during this period.

Secondly, m might get taken away from r_1 by a *BadCall* transition. In that case, n and n' disappear simultaneously from r_5 and r_6.

Thirdly, m might get taken away from r_1 by t^2 or from r_4 by t^3. If this happens then n and n' are still present on r_5 and r_6, but they have no matching m and henceforth this triple of tokens will not contribute to enabling *RemoteReturn*. Therefore, the timeout transition occurs for n' and takes away n' from place

r_6, and the *RPCFailure* transition may or may not occur for n. (We might strengthen the implementation by requiring it to occur. However as this reduces behaviour, the proof covers the strengthened case as well.)

No other cases are possible. Moreover, the cases mentioned above are mutually exclusive.

In the first case, n and n' leave no observable trace other than together with m, and the trace of m can be simulated by a similar trace through RPC leading from a *RemoteCall* to a *RemoteReturn*. In the second case, the trace of m (which is also the trace of n and n') can be simulated by a corresponding *BadCall* execution of RPC. In the third case, both m and (possibly) n leave a trace. However, due to the fact that *RemoteReturn* does not occur in this trace, it can be simulated by a trace of RPC, using the appropriate one of its *RPCFailure* transitions, i.e. t'_F if m disappeared through t^2 and t_F if m disappeared through t^3. ■6.2

Observe that the last possibility – simulating the disappearance of token m through t^3 in *RPCImpl* by executing t_F in RPC – relies crucially on the timing constraint of transition u^2 in *RPCImpl* (which implies, in particular, that it cannot occur before time δ). Suppose, on the contrary, that u^2 can occur before time δ and consider an execution in *RPCImpl* by which first u^2 and then y_1 is executed. This particular ordering of the two transitions cannot be simulated in RPC, at least not in the sense defined in section 3.3.

Note that this relies on the property that any transitions that can be synchronised with y_1 and z'_1 are not delayed. In our example, this concerns transitions *ReadCall* and *WriteCall* of *Spec*$\mathbf{i}$, *Spec*$\mathbf{i}'$ and *Spec*$\mathbf{i}u$ which satisfy this property because they do not have entry places. Any violation of this property – or, for that matter, any other hidden dependency between transitions u^2 and y_1 or z'_1 of *RPCImpl* – has an influence on claim 6.2.