A Dynamic Specification of the RPC-Memory Problem*

Egidio Astesiano – Gianna Reggio

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova, Italy
Viale Dodecaneso, 35 – Genova 16146, Italy
{ astes,reggio } @ disi.unige.it

1 Introduction

To handle the RPC-Memory Specification Problem we have used the SMoLCS method for the formal specification of dynamic systems developed by our group in the last ten years. SMoLCS consists of integrated logic specification formalisms at different levels of abstraction, from very abstract requirements till detailed designs, with methodological guidelines for supporting the development of a specification (see e.g. [2, 4, 7], for the theoretical and technical foundations and [11] for the methodological aspects).

We present in this paper the treatment of two parts of the RPC-Memory Specification Problem, corresponding to Sect. 1, 2 and 3 of [6].

The RPC-Memory Specification Problem (see [6]) could be seen as the development of a memory component (shortly denoted by MC from now on) starting with the initial requirements, corresponding to Sect. 1 and 2 of [6], later refined by a high-level implementation, whose main features are given in Sect. 3 of [6].

MC requirements We have interpreted Sect. 1 and 2 of [6] as a description of requirements, since there the activity of MC is not completely determined; e.g.: for all $n \geq 1$ the MC that receives n calls and then satisfies them in the reception order and so on, is an admissible realization.

First development step Here we have a refinement of the requirements given before; indeed Sect. 3 of [6] describes in a more precise way MC by saying to realize it using a RPC component, a reliable MC (RMC) and, as suggested afterwards also a CLERK. Moreover following the suggestion of [6] we have considered RPC apart by giving for it a reusable parameterized specification, and then we have used here a particular instantion.

Now we briefly list the main problems posed by tackling the RPC-Memory Specification Problem and how they are handled in our formalism.

The components are open dynamic systems, i.e. systems evolving along the time and interacting with the external (w.r.t. them) environment by receiving procedure calls and returning results; the idea is to model dynamic systems with labelled transition systems. Moreover we have also to handle data structures (locations, memory

* This work has been partially supported by HCM-MEDICIS, HCM-EXPRESS and MURST 40%.

values, list of arguments, ...); they are modelled by many-sorted first-order structures. The two models are integrated in the so called dynamic structures: just many-sorted first-order structures, also providing for the fact that we have to model different dynamic systems (MC, RMC, RPC, ...) characterized by different behaviours.

Then we need specifications using such models for two different purposes:

- *requirement* for expressing the starting and intermediate requirements on a dynamic system; they should determine a class of dynamic structures, all those formally and abstractly modelling systems having such requirements;
- *design* for expressing the abstract architectural design of a dynamic system; they should determine one dynamic structure, the one formally and abstractly modelling the designed system.

As for the usual logic specifications of abstract data types, a dynamic specification is a pair $(D\Sigma, AX)$, where $D\Sigma$ is a dynamic signature (a particular many-sorted signature with transition predicates) and AX is a set of axioms on $D\Sigma$.

Dynamic specifications with loose semantics (i.e. whose semantics is the class of all $D\Sigma$-dynamic structures satisfying AX) are meant for requirement. Now, first-order logic is adequate for expressing requirements on the data structures as "Init-Value cannot be a location", but cannot be used to express the requirements on the MC behaviour, for example liveness properties, like "MC must eventually issue a return for each call". Thus we have extended first-order logic with combinators of the branching-time temporal logic for expressing properties on the behaviour of the dynamic systems (see [7, 8]).

To express design specifications we need to identify essentially just one model; to this end we adopt the well-known algebraic approach of the so-called "initial semantics" (if AX is a set of positive conditional formulae, then there exists the initial model of a dynamic specification characterized by "minimal truth", i.e. a ground atomic formula holds on it iff it is a logical consequence of the axioms in AX).

The notion of implementation between classic specifications of abstract data types (see [13]) can be naturally extended to dynamic specifications and, e.g., we can formally define, and then prove, that the specification given in the first development step is a correct implementation of that of the MC requirements.

Unfortunately such proofs have to be done by hand, since at the moment there are no tools for helping the verification, except some methodological guidelines; neither automatic tools (e.g. a model checker, a theorem prover) nor theoretical ones (e.g. a sound and/or complete deductive system, a refinement calculus). We have only a software rapid prototyper for design specifications, which helps us to gain confidence in the designed system and to detect several errors, just by examining the behaviour of the specified system (see [1]).

The results of the first two steps are presented in Sect. 4 and 6 and the reusable specification for RPC is presented in Sect. 5. The basis of our specification framework, dynamic specifications, are reported in Sect. 2 and the associated development method for dynamic system is briefly sketched in Sect. 3. Finally in Sect. 7 we discuss our solution.

2 Dynamic Specifications (DSPECs)

Dynamic specifications, shortly DSPECs, extend the logic (algebraic) specification of abstract data types, see [13], to the specification of types of dynamic systems. In this paper, with the term *dynamic system* we generically denote systems that are able to modify their own state during time, so processes and concurrent/reactive/ distributed systems are typical examples.

2.1 The formal model for dynamic systems

In the DSPEC approach a dynamic system is modelled by a *labelled transition tree* defined by means of a *labelled transition system*.

Definition 1. A *labelled transition system* (shortly *lts*) is a triple $(STATE, LAB, \rightarrow)$, where $STATE$ and LAB are two sets whose elements are, respectively, the *states* and the *labels* of the system, while $\rightarrow \subseteq STATE \times LAB \times STATE$ represents the *transition relation*. A triple (st, l, st') belonging to $\rightarrow$ is called a *transition* and is usually written as $st \xrightarrow{l} st'$. □

A dynamic system D can be modelled by an lts $(STATE, LAB, \rightarrow)$ and an initial state $st_0 \in STATE$. The elements in $STATE$ that can be reached starting from st_0 are the intermediate interesting states in the life of D, while the transition relation $\rightarrow$ describes the capabilities of D to pass from one intermediate state to another one. So a transition $st \xrightarrow{l} st'$ has the following meaning: D in the state *st is able* to pass to the state st' by performing a transition whose interaction with the external world is described by the label l; thus l provides information both on the conditions on the external world making effective this capability and on the changes in the external world caused by this transition.

Given an lts, we can associate with each $st_0 \in STATE$ a *transition tree* that is a labelled tree whose root is decorated by st_0, whose nodes are decorated by states and whose edges by labels; the structure of the tree is given by means of the following rule: between two nodes decorated, respectively, by st and st' there exists an edge labelled by l iff $st \xrightarrow{l} st'$. In a transition tree the order of the edges is not meaningful and two subtrees decorated in the same way and with the same root are identified.

Concurrent dynamic systems, i.e. dynamic systems having cooperating components that are in turn other dynamic systems, can be modelled through particular ltss obtained by composing other ltss describing such components.

2.2 Dynamic structures

First, we briefly report the main definitions about first-order structures.

A *(many-sorted, first-order) signature* is a triple $\Sigma = (S, OP, PR)$, where

- S is a set (the set of the *sorts*);
- OP is a family of sets: $\{OP_{w,s}\}_{w \in S^*, s \in S}$ (*operation symbols*);
- PR is a family of sets: $\{PR_w\}_{w \in S^+}$ (*predicate symbols*).

A Σ *first-order structure* is a triple $\mathrm{A} = (\{\mathrm{A}_s\}_{s\in S}, \{Op^{\mathrm{A}}\}_{Op\in OP}, \{Pr^{\mathrm{A}}\}_{Pr\in PR})$ consisting of the *carriers*, the *interpretation of the operation symbols* and the *interpretation of the predicate symbols*. More precisely:

- if $s \in S$, then A_s is a set;
- if $Op: s_1 \times \ldots \times s_n \to s$, then $Op^{\mathrm{A}}: \mathrm{A}_{s_1} \times \ldots \times \mathrm{A}_{s_n} \to \mathrm{A}_s$ is a function;
- if $Pr: s_1 \times \ldots \times s_n$, then $Pr^{\mathrm{A}} \subseteq \mathrm{A}_{s_1} \times \ldots \times \mathrm{A}_{s_n}$.

Usually we write $Pr^{\mathrm{A}}(a_1, \ldots, a_n)$ instead of $(a_1, \ldots, a_n) \in Pr^{\mathrm{A}}$.

Given a signature Σ with set of sorts S, a *sort assignment* on Σ is an S-indexed family of sets $X = \{X_s\}_{s\in S}$.

Given a sort assignment X, the *term structure* $\mathrm{T}_\Sigma(X)$ is the Σ-structure defined as follows, using T to denote $\mathrm{T}_\Sigma(X)$:

- $x \in X_s$ implies $x \in \mathrm{T}_s$;
- $Op \in OP_{\Lambda,s}$ implies $Op \in \mathrm{T}_s$;
- $t_i \in \mathrm{T}_{s_i}$ for $i = 1, \ldots, n$ and $Op \in OP_{s_1 \ldots s_n, s}$ imply $Op(t_1, \ldots, t_n) \in \mathrm{T}_s$;
- $Op^{\mathrm{T}}(t_1, \ldots, t_n) = Op(t_1, \ldots, t_n)$ for all $Op \in OP$;
- $Pr^{\mathrm{T}} = \emptyset$ for all $Pr \in PR$.

If $X_s = \emptyset$ for all $s \in S$, then $\mathrm{T}_\Sigma(X)$ is simply written T_Σ and its elements are called *ground terms*.

In this paper we assume that structures have *nonempty carriers*.

An lts can be represented by a first-order structure A on a signature with two sorts, *state* and *label*, whose elements correspond to states and labels of the system, and a predicate $\to$: $state \times label \times state$ representing the transition relation. The triple $(\mathrm{A}_{state}, \mathrm{A}_{label}, \to^{\mathrm{A}})$ is the corresponding lts. Obviously we can have ltss whose states are built by states of other ltss (for modelling concurrent dynamic systems); in such a case we use structures with different sorts corresponding to states and labels and with different predicates corresponding to transition relations.

In a formal model for dynamic systems we may need to consider data too (for example, the data manipulated by the dynamic systems such as natural numbers); to handle these cases our structures may have also sorts that just correspond to data and not to states or labels of ltss.

The first-order structures corresponding to ltss are called *dynamic structures* and are formally defined as follows.

Definition 2.

- A *dynamic signature* $D\Sigma$ is a pair (Σ, DS), where:
  - $\Sigma = (STATE, OP, PR)$ is a signature;
  - $DS \subseteq STATE$ is the set of the *dynamic sorts*, i.e. sorts corresponding to dynamic systems (states of ltss);
  - for all $ds \in DS$ there exist a sort $lab\_ds \in STATE - DS$ (the sort of the labels) and a predicate $\_ \xrightarrow{\_} \_ : ds \times lab\_ds \times ds \in PR$ (the transition predicate).
- A *dynamic structure* on $D\Sigma$ (shortly a $D\Sigma$-dynamic structure) is just a Σ-first-order structure; the term structure $T_{D\Sigma}(X)$ is just $T_\Sigma(X)$, where X is a sort assignment on $D\Sigma$. $\square$

2.3 A logic for DSPECs

Having defined dynamic structures as our models for dynamic systems, we now introduce an appropriate logical formalism for expressing properties about them.

The properties can be subdivided in two classes: properties of the static data, including the data used for defining states and labels, that we briefly name "static properties"; and properties on the activity of the dynamic systems, such as liveness or safety requirements, that we briefly name "dynamic properties". While first-order logic is sufficient for static properties, for the dynamic ones we enrich it with the combinators of the past branching-time temporal logic with edge formulae, see [7, 8]. Moreover, since dynamic structures are classified depending on their signature also the formulae of the logic will be given below depending on a signature.

We give now some technical definitions on dynamic structures that will be used in the following. Let D be a $D\Sigma$-dynamic structure and ds a dynamic sort of $D\Sigma$.

$PATH(\mathrm{D}, ds)$ denotes the set of the *paths* for the dynamic systems of sort ds, i.e. the set of all sequences of transitions having form either (1) or ... or (4) below:

(1) $\quad \ldots\ d_{-2}\ l_{-2}\ d_{-1}\ l_{-1}\ d_0\ l_0\ d_1\ l_1\ d_2\ l_2\ \ldots$
(2) $\quad d_0\ l_0\ d_1\ l_1\ d_2\ l_2\ \ldots$
(3) $\quad \ldots\ d_{-2}\ l_{-2}\ d_{-1}\ l_{-1}\ d_0$
(4) $\quad d_0\ l_0\ d_1\ l_1\ d_2\ l_2\ \ldots\ d_n \qquad n \geq 0$

where for all integers i, $d_i \in \mathrm{D}_{ds}$, $l_i \in \mathrm{D}_{lab\_ds}$ and $(d_i, l_i, d_{i+1}) \in \rightarrow^{\mathrm{D}}$. Notice that both a single state d and a single transition $d\ l\ d'$ may be a path.

If σ has form either (3) or (4) is said *right-bounded*, while if it has form either (2) or (4) is said *left-bounded*.

If σ is right-bounded, then $LastS(\sigma)$ denotes the *last state* of σ; analogously if σ is left-bounded, $FirstS(\sigma)$ denotes the *first state* of σ; while if σ is left-bounded, then $FirstL(\sigma)$ denotes the *first label* of σ, if exists, i.e. if σ is not just a state.

$\sigma \in PATH(\mathrm{D}, ds)$ is *right-maximal* (*left-maximal*) iff either σ is not right-bounded (left-bounded) or there do not exist l, d' s.t. $(LastS(\sigma), l, d') \in \rightarrow^{\mathrm{D}}$ $((d', l, FirstS(\sigma)) \in \rightarrow^{\mathrm{D}})$.

A *composition operation* is defined on paths: $\sigma \cdot \sigma' =_{def}$
if $\sigma = \ldots\ d_{n-1}\ l_{n-1}\ d_n$ and $\sigma' = d_n\ l_n\ \ d_{n+1}\ l_{n+1} \ldots$ then
$\ldots\ d_{n-1}\ l_{n-1}\ d_n\ l_n\ d_{n+1}\ l_{n+1}\ \ldots$ else undefined.

A *pointed path* is a pair $\langle \sigma_p, \sigma_f \rangle$ s.t. σ_p is left-maximal and right-bounded, σ_f is right-maximal and left-bounded and $LastS(\sigma_p) = FirstS(\sigma_f)$; it represents a complete behaviour for the dynamic system in the state $LastS(\sigma_p)$ coinciding with $FirstS(\sigma_f)$, σ_p is the past part and σ_f the future part.

Definition (Formulae)

The set of formulae, *denoted by* $F(D\Sigma, X)$, *and the family of the sets of* path formulae, *denoted by* $\{PF(D\Sigma, X)_{ds}\}_{ds \in DS}$, *on* $D\Sigma = ((S, OP, PR), DS)$ *and a sort assignment* X *are defined by multiple induction as follows. For each* $s \in S$ *and* $ds \in DS$*:*

formulae

- $Pr(t_1, \ldots, t_n) \in F(D\Sigma, X) \quad Pr\colon s_1 \times \ldots \times s_n \in PR,\ t_i \in T_{D\Sigma}(X)_{s_i}, i = 1 \ldots n$

– $t_1 = t_2 \in F(D\Sigma, X)$ — $t_1, t_2 \in T_{D\Sigma}(X)_s$
– $\neg\, \phi_1, \phi_1 \supset \phi_2, \forall\, x \,.\, \phi_1 \in F(D\Sigma, X)$ — $\phi_1, \phi_2 \in F(D\Sigma, X), x \in X$
– $\triangle(t, \pi) \in F(D\Sigma, X)$ — $t \in T_{D\Sigma}(X)_{ds}, \pi \in PF(D\Sigma, X)_{ds}$

path formulae

– $[\,\lambda\, x \,.\, \phi\,] \in PF(D\Sigma, X)_{ds}$ — $x \in X_{ds},\ \phi \in F(D\Sigma, X)$
– $\langle \lambda x \,.\, \phi \rangle \in PF(D\Sigma, X)_{ds}$ — $x \in X_{lab\_ds},\ \phi \in F(D\Sigma, X)$
– $\pi_1\, \mathcal{U}\, \pi_2, \pi_1\, \mathcal{S}\, \pi_2 \in PF(D\Sigma, X)_{ds}$ — $\pi_1, \pi_2 \in PF(D\Sigma, X)_{ds}$
– $\bigcirc\, \pi \in PF(D\Sigma, X)_{ds}$ — $\pi \in PF(D\Sigma, X)_{ds}$
– $\neg\, \pi_1, \pi_1 \supset \pi_2, \forall\, x \,.\, \pi_1 \in PF(D\Sigma, X)_{ds}$ — $\pi_1, \pi_2 \in PF(D\Sigma, X)_{ds}, x \in X_s$ $\square$

The formulae of our logic include the usual ones of many-sorted first-order logic with equality; if $D\Sigma$ contains dynamic sorts, they include also formulae built with the transition predicates. Notice that path formulae are just an ingredient, though an important one, for building the temporal formulae.

The formula $\triangle(t, \pi)$ can be read as "for every path $\langle \sigma_p, \sigma_f \rangle$ pointed in the state denoted by t, the path formula π holds on $\langle \sigma_p, \sigma_f \rangle$". We anchor these formulae to states, following the ideas in [9]. The difference is that we do not model a single system but, in general, a type of systems, so there is not a single initial state but several of them, hence the need for an explicit reference to states (through terms) in the formulae built with $\triangle$.

The formula $[\,\lambda\, x \,.\, \phi\,]$ holds on the pointed path $\langle \sigma_p, \sigma_f \rangle$ whenever ϕ holds at the first state of σ_f, which is also the last state of σ_p; while the formula $\langle \lambda x \,.\, \phi \rangle$ holds on the pointed path $\langle \sigma_p, \sigma_f \rangle$ if σ_f is not just a single state and ϕ holds at the first label of σ_f.

Finally, $\bigcirc$, $\mathcal{U}$ and $\mathcal{S}$ are the so called next, (future) until and (past) since combinators.

If A is a Σ-structure, a *variable evaluation* $V \colon X \to \mathrm{A}$ is a sort-respecting assignment of values in A to *all* the variables in X. If $t \in \mathrm{T}_\Sigma(X)$, the *interpretation of* t *in* A *w.r.t.* V is denoted by $t^{\mathrm{A},V}$ and given as follows:
– $x^{\mathrm{A},V} = V(x)$
– $Op(t_1, \ldots, t_n)^{\mathrm{A},V} = Op^{\mathrm{A}}(t_1^{\mathrm{A},V}, \ldots, t_n^{\mathrm{A},V})$.

Definition (Semantics of formulae) *Let* D *be a* $D\Sigma$*-dynamic structure and* V *a variable evaluation of* X *in* D*; then we define by multiple induction:*
– *the validity of a formula* ϕ *in* D *w.r.t.* V *(written* $\mathrm{D}, V \models \phi$*),*
– *the validity of a path formula* π *on a pointed path* $\langle \sigma_p, \sigma_f \rangle$ *in* D *w.r.t.* V *(written* $\mathrm{D}, V, \langle \sigma_p, \sigma_f \rangle \models \pi$*),*
as follows:
formulae

– $\mathrm{D}, V \models Pr(t_1, \ldots, t_n)$ *iff* $(t_1^{\mathrm{D},V}, \ldots, t_n^{\mathrm{D},V}) \in Pr^{\mathrm{D}}$
– $\mathrm{D}, V \models t_1 = t_2$ *iff* $t_1^{\mathrm{D},V} = t_2^{\mathrm{D},V}$
– $\mathrm{D}, V \models \neg\, \phi$ *iff* $\mathrm{D}, V \not\models \phi$
– $\mathrm{D}, V \models \phi_1 \supset \phi_2$ *iff either* $\mathrm{D}, V \not\models \phi_1$ *or* $\mathrm{D}, V \models \phi_2$
– $\mathrm{D}, V \models \forall\, x \,.\, \phi$ *iff for each* $v \in \mathrm{D}_s$*, with* s *sort of* x*,* $\mathrm{D}, V[v/x] \models \phi$

– $D, V \models \triangle(t, \pi)$ *iff*
for each $\langle \sigma_p, \sigma_f \rangle$ *s.t.* $FirstS(\sigma_f) = t^{D,V}$, $D, V, \langle \sigma_p, \sigma_f \rangle \models \pi$

path formulae

– $D, V, \langle \sigma_p, \sigma_f \rangle \models [\, \lambda\, x \,.\, \phi \,]$ *iff* $D, V[FirstS(\sigma_f)/x] \models \phi$
– $D, V, \langle \sigma_p, \sigma_f \rangle \models \langle \lambda x \,.\, \phi \rangle$ *iff* $FirstL(\sigma_f)$ *is defined and* $D, V[FirstL(\sigma_f)/x] \models \phi$.
– $D, V, \langle \sigma_p, \sigma_f \rangle \models \pi_1 \,\mathcal{U}\, \pi_2$ *iff*
there exist σ_1, σ_2 *s.t.* $\sigma_f = \sigma_1 \cdot \sigma_2$, $D, V, \langle \sigma_p \cdot \sigma_1, \sigma_2 \rangle \models \pi_2$ *and*
for each σ_1', σ_1'' *s.t.* $\sigma_1 = \sigma_1' \cdot \sigma_1''$ *and* $\sigma_1' \neq \sigma_1$, $D, V, \langle \sigma_p \cdot \sigma_1', \sigma_1'' \cdot \sigma_2 \rangle \models \pi_1$
– $D, V, \langle \sigma_p, \sigma_f \rangle \models \pi_1 \,\mathcal{S}\, \pi_2$ *iff*
there exist σ_1, σ_2 *s.t.* $\sigma_p = \sigma_1 \cdot \sigma_2$, $D, V, \langle \sigma_1, \sigma_2 \cdot \sigma_f \rangle \models \pi_2$ *and*
for each σ_2', σ_2'' *s.t.* $\sigma_2 = \sigma_2' \cdot \sigma_2''$ *and* $\sigma_2'' \neq \sigma_2$, $D, V, \langle \sigma_1 \cdot \sigma_2', \sigma_2'' \cdot \sigma_f \rangle \models \pi_1$
– $D, V, \langle \sigma_p, \sigma_f \rangle \models \bigcirc\, \pi$ *iff* $\sigma_f = st\ l\ \sigma'$ *and* $D, V, \langle \sigma_p \cdot (st\ l\ FirstS(\sigma')), \sigma' \rangle \models \pi$
– $D, V, \langle \sigma_p, \sigma_f \rangle \models\ \neg\, \pi$ *iff* $D, V, \langle \sigma_p, \sigma_f \rangle \not\models \pi$
– $D, V, \langle \sigma_p, \sigma_f \rangle \models \pi_1 \supset \pi_2$ *iff either* $D, V, \langle \sigma_p, \sigma_f \rangle \not\models \pi_1$ *or* $D, V, \langle \sigma_p, \sigma_f \rangle \models \pi_2$
– $D, V, \langle \sigma_p, \sigma_f \rangle \models \forall\, x \,.\, \pi$ *iff for each* $v \in D_{ds}$, $D, V[v/x], \langle \sigma_p, \sigma_f \rangle \models \pi$

ϕ *is* valid *in* D *(written* $D \models \phi$*)* *iff* $D, V \models \phi$ *for all evaluations* V. □

In the above definitions we have used a minimal set of combinators; in practice, however, it is convenient to use other, derived, combinators; we list below those that we shall use in this paper, together with their semantics.

– **true**, **false**, $\vee$, $\wedge$, $\exists$ and $\equiv$, defined in the usual way
– $\Diamond\, \pi =_{\text{def}}$ **true** $\mathcal{U}\, \pi$ (eventually in the future π)
$D, V, \langle \sigma_p, \sigma_f \rangle \models \Diamond\, \pi$ iff
there exist σ_1, σ_2 s.t. $\sigma_f = \sigma_1 \cdot \sigma_2$, and $D, V, \langle \sigma_p \cdot \sigma_1, \sigma_2 \rangle \models \pi$
– $\Diamond\mathbf{P}\, \pi =_{\text{def}}$ **true** $\mathcal{S}\, \pi$ (some time in the past π)
$D, V, \langle \sigma_p, \sigma_f \rangle \models \Diamond\mathbf{P}\, \pi$ iff
there exist σ_1, σ_2 s.t. $\sigma_p = \sigma_1 \cdot \sigma_2$, $D, V, \langle \sigma_1, \sigma_2 \cdot \sigma_f \rangle \models \pi$
– $\bigtriangledown(t, \pi) =_{\text{def}}\ \neg\ \triangle\,(t,\ \neg\, \pi)$ (in one case)
$D, V \models \bigtriangledown(t, \pi)$ iff
there exists $\langle \sigma_p, \sigma_f \rangle$ s.t. $FirstS(\sigma_f) = t^{D,V}$ and $D, V, \langle \sigma_p, \sigma_f \rangle \models \pi$

Whenever in ϕ there are no free variables of dynamic sort except x: $[\, \lambda\, x \,.\, \phi \,]$ is abbreviated to $[\, \phi \,]$, moreover $[\, s = t \,]$ is abbreviated to $[\, t \,]$; analogously $\langle \lambda x \,.\, \phi \rangle$ and $\langle l = t \rangle$ are abbreviated respectively to $\langle \phi \rangle$ and $\langle t \rangle$.

2.4 Dynamic specifications

Definition 3. A *DSPEC (dynamic specification)* is a pair $SP = (D\Sigma, AX)$ where $D\Sigma$ is a dynamic signature and $AX \subseteq F(D\Sigma, X)$.

The *models* of SP, $Mod(SP)$, are the $D\Sigma$-dynamic structures D s.t. the formulae in AX are valid in D. □

We need to consider two different kinds of DSPECs:

requirement for expressing the starting and intermediate requirements of a dynamic system; a requirement DSPEC should determine a class of dynamic structures, all those formally and abstractly modelling systems having such requirements; technically the semantics of a requirement DSPEC is the class of its models (loose semantics);

design for expressing the abstract design of a dynamic system, i.e. to abstractly and formally define the way we intend to design the system; a design DSPEC should determine one dynamic structure, the one formally and abstractly modelling the designed system; technically the semantics of a design DSPEC is the initial element in the class of its models, if any (recall that the initial element is unique up to isomorphism).

A DSPEC may not have an initial model, since it might contain an axiom like $t_1 = t_2 \ \vee \ t_1 = t_3$; so we have to restrict the form of the axioms used in design specifications, by considering only *conditional* axioms having the following form: $\alpha_1 \ \wedge \ \ldots \ \wedge \ \alpha_n \ \supset \ \alpha$, where α_i and α are atoms i.e. either $Pr(t_1, \ldots, t_n)$ or $t = t'$.

Proposition 4. *Given a DSPEC* $SP = (D\Sigma, AX)$ *where* AX *is a set of conditional axioms, then there exists (unique up to isomorphism)* I_{SP} *initial in* $Mod(SP)$ *characterized by*

- *for all* $t_1, t_2 \in \mathrm{T}_{D\Sigma}$ *of the same sort* $\quad \mathrm{I}_{SP} \models t_1 = t_2$ *iff* $AX \vdash t_1 = t_2$;
- *for all* $Pr \in PR$ *and all* $t_1, \ldots, t_n \in \mathrm{T}_{D\Sigma}$ *of appropriate sorts* $\mathrm{I}_{SP} \models Pr(t_1, \ldots, t_n)$ *iff* $AX \vdash Pr(t_1, \ldots, t_n)$;

where $\vdash$ *denotes provability in the Birkhoff sound and complete deductive system for conditional axioms, whose rules are:*

$$t = t \qquad \frac{t = t'}{t' = t} \qquad \frac{t = t' \quad t' = t''}{t = t''}$$

$$\frac{t_i = t'_i \quad i = 1, \ldots, n}{Op(t_1, \ldots, t_n) = Op(t'_1, \ldots, t'_n)} \qquad \frac{t_i = t'_i \quad i = 1, \ldots, n \quad Pr(t_1, \ldots, t_n)}{Pr(t'_1, \ldots, t'_n)}$$

$$\frac{\alpha_1 \ \wedge \ \ldots \ \wedge \ \alpha_n \ \supset \ \alpha \qquad \alpha_i \quad i = 1, \ldots, n}{\alpha}$$

$$\frac{F}{\overline{V}(F)} \quad V : X \to \mathrm{T}_{\Sigma}(X)$$

$\overline{V}(F)$ *is* F *where each occurrence of a variable, say* x*, has been replaced by* $V(x)$. □

A notion of "correct implementation" between DSPECs has been given (see [4]) as follows: a requirement SP is *implemented* by SP' with respect to $\mathcal{F}$, a function from specifications into specifications, iff $Mod(\mathcal{F}(SP')) \subseteq Mod(SP)$. The function $\mathcal{F}$ describes how the parts of SP are realized in SP' (implementation as realization); while implementation as refinement is obtained by requiring inclusion of the classes of models. Notice that this definition applies whatever the kind of SP' (either requirement or design); in the latter case the class of it models just contains the initial one and those isomorphic to it.

3 A Development Process for Dynamic Systems

3.1 Development process

The development process supported by DSPEC consists of different phases from the presentation of the informal idea of what we have to realize till the coding of the dynamic system in a programming language. From a phase it is possible to go back to the previous phases either because some part has been modified or because an error has been found.

Each phase of the development process is characterized by the production of a document, having a particular structure, which guides and documents the activities of the phase.

The development process starts from what we call *natural description*, i.e. a document, given by the client, describing the system to be realized using some natural language.

PHASE 1 During this phase we analyse the dynamic system trying to determine its essential requirements and after to specifiy them by a requirement DSPEC.
The analysis starts from the natural description; but it may be that in the natural description there are ambiguities, inconsistencies, or parts whose only possible interpretation is not sensible; these points, called *shadow spots*, should be reported in a document together with possible choices; if the shadow spots are too many or too relevant preventing to determine the requirements, the natural description should be returned to the client to be modified.
The *border determination* is another part of document of this phase; it gives the motivations for deciding which parts of the dynamic system have been included in the specification. That is relevant, because within DSPECs it is not possible to give requirements on the environment of the specified dynamic system; thus where to place the border of the dynamic system, i.e. which parts of the outside environment to specify, depends on the relevance of the requirements about such parts.
Summarizing, during PHASE 1 we have to produce a document consisting of: the natural description, the border determination, the shadow spots description and the specification of the requirements, just a requirement DSPEC.

PHASE 2 The requirement DSPEC given in PHASE 1 is developed through an appropriate number of (development) steps; in each step we make more detailed the features of the system to realize until its complete definition.
A development step is further split in:
- an *analysis step*, in which we either refine the requirement on the system or design some of its parts; the result of the analysis is formalized by a DSPEC, which may be either of kind requirement (when the requirements have been refined), or design (when each part of the system has been designed), or mixed, i.e. a combination of subspecifications of kind design with other of kind requirement (when only some parts of the system have been designed).
- a *correctness verification step* which verifies that the specification given at this step is a correct implementation of the previous one (for the first step of this phase, of the requirement specification of PHASE 1). Clearly, it may

happen that the current specification is wrong (i.e. cannot be a correct implementation), in such cases we have either to redo the current step, or if instead the error is in the previous steps or in the requirements, to go back to modify them appropriately.

Summarizing, during a development step of PHASE 2 we have to produce a document consisting of: a natural description saying which choices have been done in the step, a DSPEC of the appropriate kind and a *justification of the correctness of the step.*

PHASE 2 ends with a step producing a design DSPEC.

PHASE 3 If the specification given in the last development step of PHASE 2 is prototypable, then we can perform tests on it using the prototyping tool (see [1]) for verifying the consistency between the starting idea of the system (in the natural description) and the produced design. The rapid prototyper given a design specification of a dynamic system and a state of such system generates in an interactive way (parts of) the labelled transition tree starting from such state, and so allows to analyse the behaviour of the system starting from such state.

If the result of such tests is not satisfactory, we have go back to modify the previous steps.

PHASE 4 If the result of the tests of PHASE 3 is satisfactory, the dynamic system is coded using a programming language following the usual criteria of efficiency and correctness.

3.2 How to write a DSPEC

The development process presented in the previous subsection requires to write down several DSPECs; here we present some guidelines for writing a DSPEC of a dynamic system.

First, the dynamic system should be classified in *simple* or *concurrent*; concurrent systems are those whose activity is determined by the activity of several components, which are in turn dynamic systems (either simple or concurrent), also of different types. Thus the guidelines below distinguish between the two cases. Clearly the classification in simple and concurrent of a system is relative to a step and a simple system may be refined by a concurrent one in the next step.

Simple dynamic system

Basic data structures Determine and specify the data structures used by the dynamic system.

States Determine and specify the intermediate relevant situations in the life of the dynamic system (i.e. the states of the lts modelling it) as a data structure, with a main sort corresponding to such states. Then the main sort is made dynamic.

Interactions Determine and specify the interactions of the dynamic system with the external world (i.e. the labels of the lts modelling it) as a data structure, whose main sort will be the sort of label associated with the dynamic sort given in the previous point, and having an operation for each *kind* of interactions; these operations may have several arguments describing the information exchanged between the system and the external world during each interaction of such kind.

Activity Determine and specify the activity of the dynamic system (i.e. the transitions of the lts modelling it) represented by the arrow predicate associated with the dynamic sort given before.

requirement We have to give the relevant properties on the transitions, which are usually grouped depending on the kind of the performed interaction.
Let $L: s_1 \times \ldots \times s_n \to lab\_ds$ be an operation representing the interactions of a certain kind; these properties may express:

- *necessary conditions* on the initial/final state of a capability with interaction of " kind L" (which reactions we expect from the system after its execution and what must have happened before):
$st \xrightarrow{L(t_1,\ldots,t_n)} st' \supset \phi(st, t_1, \ldots, t_n, st') \wedge \triangle(st', \pi_f) \wedge \triangle(st, \pi_p)$
where $t_1, \ldots, t_n$ are terms of sort $s_1, \ldots, s_n$ respectively, and st, st' are terms of sort ds, π_f is a path formula built with the future temporal combinators and π_p with the past ones.
- *sufficient conditions* for the system to have a transition capability with interaction of "kind L"; these properties are very important in the specification of a reactive system, since they express that the system in some cases eventually must be able to accept a certain external stimulation. These properties may have various forms:
  - $\phi(st) \supset \nabla(st, \langle L(t_1, \ldots, t_n)\rangle)$
(the system has (immediately) a capability to do such interaction)
  - $\phi(st) \supset \triangle(st, \Diamond \langle L(t_1, \ldots, t_n)\rangle)$
(in any case the system will eventually perform such interaction)
  - $\phi(st) \supset \triangle(st, \Diamond [\lambda x . \nabla (x, \langle L(t_1, \ldots, t_n)\rangle)])$
(in any case the system will eventually have the capability to do such interaction)

  where $t_1, \ldots, t_n$ are terms of sort $s_1, \ldots, s_n$ respectively, and st is a term of sort ds.

Whole behaviour properties i.e. properties that are not related to the occurrence of particular interactions but refer to a whole behaviour. Their structure is simply $\triangle(st, \pi)$, where st is a term of dynamic sort; e.g. π may express a *responsiveness* property, i.e. has form $(\Box \Diamond \pi_1) \supset (\Box \Diamond \pi_2)$.

design We have to define the system transitions by giving a set of conditional axioms of the form: $cond(st, l, st') \supset st \xrightarrow{l} st'$, where st, st', l are terms of sort ds and lab_ds respectively, and $cond(st, l, st')$ is a conjunction of atoms; recall that the only transitions of the specified system are those which can be proved by the given axioms by using the deductive system of Prop. 4.

Concurrent dynamic system

Basic data structures As in the simple case.

Components Determine and specify following these same guidelines the components of the concurrent system, which are in turn dynamic systems.

States As in the simple case, but now they are defined by putting together the states of the components.

Interactions As in the simple case.

Activity Determine and specify the activity of the dynamic system, the transitions, by considering also the activities (transitions) of its components.

requirement As for the simple case, the only difference is that not only the system can perform transitions, but also its components.

design We have to define the system transitions by giving a set of conditional axioms of the form:

$$\phi(l, l_1, \ldots, l_n) \ \wedge \ c_1 \xrightarrow{l_1} c'_1 \ \wedge \ \ldots c_n \xrightarrow{l_n} c'_n \ \supset$$
$$st(c_1, \ldots, c_n) \xrightarrow{l} st'(c'_1, \ldots, c'_n),$$

where $l_1, \ldots, l_n, l$,are terms of label sorts, $c_1, \ldots, c_n, c'_1, \ldots, c'_n$ are variables of sort states of the components and $st(c_1, \ldots, c_n)$ $(st'(c_1, \ldots, c_n))$ is a term denoting a state of the concurrent system where the components are in the states $c_1, \ldots, c_n$ respectively. Notice that the above constraints on the form of axioms ensure that only the interactions of the component transitions are relevant for the composition, and so that interactions really represent the transition interfaces.

3.3 Specification presentation

To be able to present in a sensible way the specifications produced during the development process, a specification language for DSPEC, METAL see [10], has been developed with a precise "friendly" syntax (e.g. no esoteric symbols but mnemonic keywords) and with facilities for helping to write down complex and large specifications. The constructs of METAL will be briefly explained when used in the specifications of the RPC-Memory Specification Problem, by comments enclosed by [and].

Furthermore, each formal DSPEC is accompanied by a strictly correspondent informal specification (natural language text following a particular structure). In this paper these informal specifications are presented as line-by-line comments of the formal ones, but it is also possible to give as first those informal specifications, present and discuss them with the client and after derive from them the formal counterpart, see [3]. Notice the strict correspondence between the informal text and the corresponding formal part.

It is also possible to associate a graphical presentation, both with the formal and the informal specifications, to improve their readability; to give the flavour we report some of the diagrams associated with RPC-Memory Specification Problem, see e.g. Fig. 1.

4 MC Requirement Specification (PHASE 1)

To give the requirements on MC corresponds to PHASE 1 of the development process associated with DSPECs (see Sect. 3), and means to determine which are all processes described by Sect. 1 and 2 of [6]; i.e. to define the class of all dynamic structures modelling such processes by using a requirement DSPEC.

Below we report the documents produced during this phase together with some comments. The formal specifications are written using the specification language

METAL, see [10], while the corresponding informal specifications are reported as line-by-line comments and typed using the *italic font*.

Natural description Sect. 1 and 2 of [6].

Border determination The universe of MC consists of MC itself and of the other components; moreover within a component there are several processes in parallel calling the MC's procedures. In the natural description there is a requirement on such processes: they cannot start a new procedure call before to have terminated any previous ones (by receiving a result). Since this property will be used to define MC, the border of the specified dynamic system should enclose all components; however in this report to save space we consider it to enclose exactly MC.

Basic data structures The basic data structures part contains the specifications of those data structures used by the dynamic system.

Universe of the values [METAL offers a textual notation for all signature items (sorts, operations, predicates and axioms) plus a special notation for constants (**cn**), which are the zero-ary operations; "if ... then ..." is just the syntax for the conditional logic combinator.]

UNIVERSE =
universe of the values either arguments or return values of the MC *procedures*
requirement
sort universe

cn Init_Value: universe *value initially contained in the locations*
cn Write_End: universe *value returned by a successful Write call*
cn BadArg, MemFailure: universe *exceptional values*

BadArg and MemFailure are different
ax not BadArg = MemFailure

checks whether a universe element is a memory value/a location
pr Is_MemVal, Is_Loc: universe

a universe element cannot be both a memory value and a location
ax not (Is_MemVal(u) and Is_Loc(u))

Init_Value is a memory value while Write_End is not so
ax Is_MemVal(Init_Value) and not Is_MemVal(Write_End)

BadArg and MemFailure are neither memory values nor locations
ax not (Is_MemVal(BadArg) or Is_MemVal(MemFailure))
ax not (Is_Loc(BadArg) or Is_Loc(MemFailure))
end

Process identifier

PID =
identifiers of the processes (in some other components) originally issuing the calls
requirement sort pid **end**

Calls ["**use**" is the METAL construct for modularly building specifications; below **use** UNIVERSE means that the CALL specification has all sorts, operations and predicates of UNIVERSE with the properties expressed in UNIVERSE.]

CALL = *calls of the* MC *procedures*
design
use UNIVERSE
sort call

takes two universe elements and returns a Write call
op Write: universe universe -> call
takes a universe element and returns a Read call
op Read: universe -> call

pr Correct: call *checks if a call is correct*
if the first argument is a location and the second a memory value, then a Write call is correct
ax if Is_Loc(u) and Is_MemVal(u′) then Correct(Write(u,u′))
if the argument is a location, then a Read call is correct
ax if Is_Loc(u) then Correct(Read(u))
end

Interactions

MC_INTERACT =
requirement
use CALL, PID

to receive a call and a process identifier (of the process in some other component that has originally issued the call)
op RECEIVE: call pid -> lab_mc

to return a result and a process identifier (of the process in some other component that finally will get the result)
op RETURN: universe pid -> lab_mc

auxiliary predicate
checks if an interaction does not concern a given process identifier
pr No_Concern: lab_mc pid
ax No_Concern(RECEIVE(c,pi),pi′) iff not pi = pi′
ax No_Concern(RETURN(u,pi),pi′) iff not pi = pi′
end

Notice that here we have used a requirement DSPEC for the MC interactions, since we only know that MC has at least the above interactions, but we do not know completely such interactions.

States [The METAL construct "**dsort** mc: _ – _ –> _" declares "mc" to be a dynamic sort and implicitly also the associated sort of labels ("lab_mc") and transition predicate "**pr** _ – _ –> _: mc lab_mc mc".]

MC_STATE =
requirement
use UNIVERSE
dsort mc: _ – _ –> _

returns the content of a location in a given state
op Cont: mc universe -> universe
the content of a location is a memory value
ax if Is_Loc(u) then Is_MemVal(Cont(mc,u))
end

Activity [Below "…in any case …", "until", "since" and "next" are the METAL syntaxes for the combinators of the temporal logic $\triangle(\ldots,\ldots)$, $\mathcal{U}$, $\mathcal{S}$ and $\bigcirc$, introduced in Sect. 2.3.]

MC =
requirement
use MC_INTERACT, MC_STATE

*** 1 ***
if MC *receives a non-correct call and pi, then*
ax if not Correct(c) and mc – RECEIVE(c,pi) –> mc' then
    in any case it will perform interactions non-concerning pi until
    mc' in any case < No_Concern(y,pi) > until
        will return either BadArg or MemFailure and pi
        (u = BadArg or u = MemFailure) and < RETURN(u,pi) >

*** 2 ***
if MC *receives a correct call Read(u) and pi, then*
ax if mc – RECEIVE(Read(u),pi) –> mc' and Correct(Read(u)) then
    in any case it will perform interactions non-concerning pi until
    mc' in any case < No_Concern(y,pi) > until
        either will return the content of u and pi
        ((exists u': [Cont(x,u) = u'] and next < RETURN(u',pi) >)
        or will return MemFailure and pi
        or < RETURN(MemFailure,pi) >)

*** 3 ***
if MC *receives a correct call Write(u,u′) and pi, then*
ax if mc – RECEIVE(Write(u,u′),pi) –> mc′ and Correct(Write(u,u′)) then
in any case it will perform interactions non-concerning pi until
mc′ in any case < No_Concern(y,pi) > until
either will change the content of u to u′ and after
([Cont(x,u) = u′] and next
will perform interactions non-concerning pi until
(< No_Concern(y,pi) > until
will return Write_End and pi
< RETURN(Write_End,pi) >))
or will return MemFailure and pi
or < RETURN(MemFailure,pi) >

*** 4 ***
if MC *returns BadArg and pi, then*
ax if mc – RETURN(BadArg,pi) –> mc′ then
in any case it has performed interactions non-concerning pi since
mc′ in any case < No_Concern(y,pi) > since
has received a non-correct call and pi
< exists c: not Correct(c) and y = RECEIVE(c,pi) >

*** 5 ***
if MC *returns MemFailure and pi, then*
ax if mc – RETURN(MemFailure,pi) –> mc′ then
in any case it has performed interactions non-concerning pi since
mc′ in any case < No_Concern(y,pi) > since
has received a call and pi
< exists c: y = RECEIVE(c,pi) >

*** 6 ***
if MC *returns Write_End and pi, then*
ax if mc – RETURN(Write_End,pi) –> mc′ then
in any case it has performed interactions non-concerning pi since
mc′ in any case < No_Concern(y,pi) > since
has received a correct write call and pi
< exists u, u′: Correct(Write(u,u′)) and >
y = RECEIVE(Write(u,u′),pi)

*** 7 ***
if MC *returns a memory value and pi, then*
ax if mc – RETURN(u,pi) –> mc′ and Is_MemVal(u) then
in any case it has performed interactions non-concerning pi since
mc′ in any case < No_Concern(y,pi) > since
has received a correct read call and pi
< exists u′: Correct(Read(u)) and y = RECEIVE(Read(u),pi) >

*** 8 ***

if MC *changes the content of a location u to u′, then*

ax if mc – y –> mc′ and (not Cont(mc,u) = u′) and Cont(mc′,u) = u′ then

in any case for some pi

mc′ in any case exists pi:

it has performed interactions non-concerning pi since

(< No_Concern(y,pi) > since

has received a correct call Write(u,u′) and pi

< y = RECEIVE(Write(u,u′),pi) and Correct(Write(u,u′)) >)

end

For MC and any other dynamic system appearing in the RPC-Memory Specification Problem, together with the obvious properties about reacting to stimuli, as 1, 2 and 3, we have considered also a set of properties which may be termed "no unsolicited reactions", as 4, ..., 8 (i.e. properties requiring that some activities of a system can be present only as reactions to previously received stimuli).

Shadow spots The points listed below are not very clear in the natural description of MC; for each of them we report also how we have settled it.

- Which are the available locations in a memory component? We have chosen that all locations are available.
- Are properties 4, ..., 7 necessary? We have opted for yes.
- Are the locations subject to failures?, i.e. can they change their content by themselves? We have opted for no (property 8).
- Can MemFailure be returned as result of a non-correct call? We have opted for yes.

The structure of the specification of the MC requirements is graphically reported in Fig. 1; there square boxes correspond to specifications of data structures and rounded boxes to dynamic systems; enclosure of boxes corresponds to the fact that a specification uses another specification. The small letters enclosed by parenthesis near data structure names are used to denote generic elements of such structures. Finally the arrows leaving the MC box correspond to the MC interactions; "..." means that MC may have other interactions

5 The Reusable Specification of RPC

In this section we give the specification of the parametric simple system corresponding to RPC, to be (re)used in the specification of the implementation of MC. From Sect. 3 of [6] we have that the parameter is a data structure corresponding to the procedure names; for simplicity, in the following we take instead as parameter a data structure corresponding to the universe of values, the process identifiers, the calls and the remote calls.

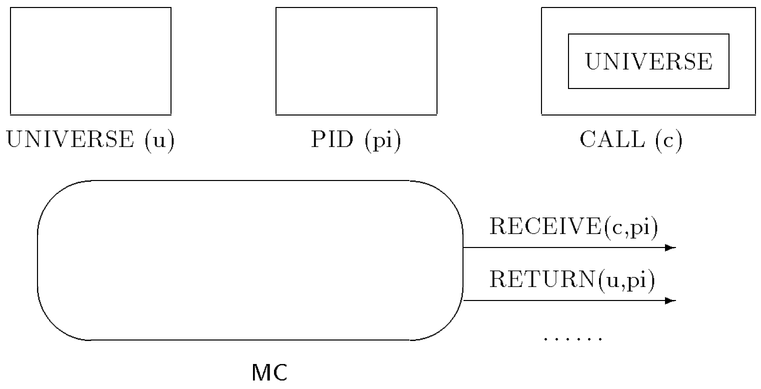

Fig. 1. The requirement specification of MC

The acceptable actual parameters Clearly not any data structure may be used to instantiate RPC; the properties on the acceptable ones are given by the following specification. An actual parameter is acceptable iff its signature contains all items of the signature of PAR with the properties expressed by the axioms of PAR.

PAR = *the essential properties of the* RPC *parameter*
requirement
use NAT
universe of the values either arguments or results of the RPC *procedures*
sort universe

sort exception *exceptional values*
cn BadCall, RPCFailure: exception

sort proc *procedure names*
given a procedure name returns the number of its arguments
op ArgNum: proc -> nat

an exception and a procedure name are universe values
op E: exception -> universe
op P: proc -> universe

check if a universe element is a list of length n
pr Is_List: universe nat

if a universe element is a list with length n and with length m, then n is equal to m
ax if Is_List(u,n) and Is_List(u,m) then n = m

exceptions and procedure names are not list
ax not Is_List(E(e),n) and not Is_List(P(p),n)

identifiers of the processes (in some other components) originally issuing the calls
sort pid

sort rcall   *remote calls received by* RPC
the remote calls are made by two universe elements (the procedure name and the list of the arguments)
op < _ ; _ >: universe universe -> rcall
ax exists u, u′: rc = < u ; u′ >
ax if < u ; u′ > = < u1 ; u1′ > then u = u1 and u′ = u1′

checks if a remote call is correct
pr Remote_Correct: rcall

a remote call is correct iff
ax Remote_Correct(u,u′) iff
u is a procedure name p and u′ a list whose length is the number of arguments of p
exists p: u = P(p) and Is_List(u′, ArgNum(p))

sort call   *calls of the* MC *procedures*

transforms a remote call into the corresponding call
op Call: rcall -> call
end

Interactions The interactions of the reusable RPC are given by a parametric specification. [**generic** is the keyword for introducing the formal parameter X (a specification), while PAR expresses which are the correct actual ones.]

RPC_P_INTERACT =
generic X: PAR
design
use X, PID

to receive (from the sender) a remote call and an identifier (of the process in some other component that has originally issued the call)
op RECEIVE_REM: rcall pid -> lab_rpc

to return (to the sender) a result (of a remote call) and an identifier (of the process in some other component that finally will get the result)
op RETURN_REM: universe pid -> lab_rpc

to send a call (to the receiver) and an identifier (of the process in some other component that has originally issued the call)
op SEND: call pid -> lab_rpc

to receive a (call) result (from the receiver) and an identifier (of the process in some other component that finally will get the result)
op RECEIVE_CALL_RES: universe pid -> lab_rpc
end

States We have no requirements on the states of the reusable RPC.

RPC_P_STATE = **requirement dsort** rpc: _ – _ –> _ **end**

Activity In this case we have also a kind of properties, that may be termed "acceptance vitality", as R7 and R8 (i.e. properties requiring that the system must surely, at certain points, have the capabilities to receive stimuli; these properties e.g. avoids that the forever stopped system is a correct implementation.
[Below "eventually" is the METAL syntax for the combinator of the temporal logic $\Diamond$, introduced in Sect. 2.3.]

RPC_P =
generic X: PAR
design
use RPC_P_INTERACT(X), RPC_P_STATE

*** R1 ***
if RPC *receives a correct remote call rc and pi, then*
ax if rpc – RECEIVE_REM(rc,pi) –> rpc' and Remote_Correct(rc) then
 in any case it will eventually
 rpc' in any case eventually
  either send the call corresponding to rc and pi
  (< SEND(Call(rc),pi) >
  or return RPCFailure and pi
  or < RETURN_REM(E(RPCFailure),pi) >)

*** R2 ***
if RPC *receives a non-correct remote call rc and pi, then*
ax if rpc – RECEIVE_REM(rc,pi) –> rpc' and not Remote_Correct(rc) then
 in any case it will eventually return BadCall and pi
 rpc' in any case eventually < RETURN_REM(E(BadCall),pi) >

*** R3 ***
if RPC *receives a result u and pi, then*
ax if rpc – RECEIVE_CALL_RES(u,pi) –> rpc' then
 in any case it will eventually
 rpc' in any case eventually
  return either u or RPCFailure and pi
  ((u′ = u or u′ = E(RPCFailure)) and < RETURN_REM(u′,pi) >)

```
**  R4  **
if RPC sends a call c and pi, then
ax  if rpc – SEND(c,pi) –> rpc' then
        in any case it
        rpc' in any case
            has not sent any other calls with pi since
            < not exists c': y = SEND(c',pi) > since
                has received a remote call corresponding to c and pi
                < exists rc: c = Call(rc) and y = RECEIVE_REM(rc,pi) >

**  R5  **
if RPC returns a result u different from RPCFailure and pi, then
ax  if rpc – RETURN_REM(u,pi) –> rpc' and not(u = RPCFailure) then
        in any case it
        rpc' in any case
            has not returned any other results with pi since
            < not exists u': y = RETURN_REM(u',pi) > since
                has received u and pi
                < RECEIVE_CALL_RES(u,pi) >

**  R6  **
if RPC returns RPCFailure and pi, then
ax  if rpc – RETURN_REM(RPCFailure,pi) –> rpc' then
        in any case it
        rpc' in any case
            has not returned any other results with pi since
            < not exists u': y = RETURN_REM(u',pi) > since
                (either has received some result and pi
                ( < exists u': y = RECEIVE_CALL_RES(u',pi) >
                or has received some remote call and pi)
                or < exists rc: y = RECEIVE_REM(rc,pi) > )

**  R7  **
in any case RPC will eventually
ax  rpc in any case eventually
        reach a state where may receive any remote call
        [ forall rc, pi: exists rpc': x – RECEIVE_REM(rc,pi) –> rpc' ]

**  R8  **
in any case RPC will eventually
ax  rpc in any case eventually
        reach a state where may receive any result
        [forall u, pi: exists rpc': x – RECEIVE_CALL_RES(u,pi) –> rpc']
end
```

6 MC Development (First Step of PHASE 2)

Here we consider a refinement of the requirements on MC, given in Sect. 4, following what said in Sect. 3 of [6]; there MC is implemented by using a RPC component, a reliable MC (RMC) and, as suggested afterwards, also a CLERK. Moreover, always following [6], the RPC component is given by instantiating a reusable process, whose parameterized specification has been given apart in Sect. 5. Notice that in this development step a simple system has been refined into a concurrent one.

As said in Sect. 3 a development step should produce, other than a specification, a natural description of the performed development and a justification of the correctness of such step; furthermore the specification should be structured following the system structure, as graphically reported in Fig. 2. There the components are represented by enclosing the rounded boxes corresponding to them into that of the whole system.

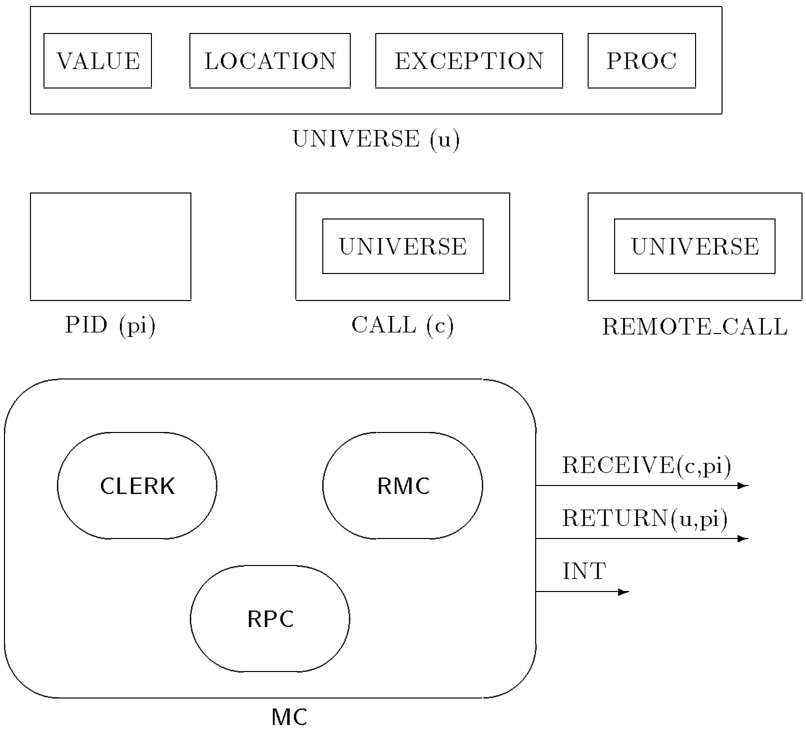

Fig. 2. The requirement specification of MC (First Step of PHASE 2)

Natural description

Sect. 3 of [6] plus the following assumptions on which are the processes composing the implementing system.

The implementation consists of three processes: RMC, RPC and CLERK handling the communication with outside MC. CLERK receives from outside the calls of the MC procedures and after forwards them to RPC. Furthermore it receives the results of the calls from RPC; if such results are different from RPCFailure, then after it will return them outside MC, otherwise either it retries to send to RPC the call that has originated the result or it will return a MemFailure exception; surely for each call it will return a result eventually.

The assumption on the activity of the processes calling the MC procedures "they cannot start a new procedure call before to have terminated any previous ones (by receiving a result)" is used to give this implementation of MC, since it implies that it is not possible to have two outstanding calls issued by the same process; so it allows to use the process identifier to uniquely identify the outstanding calls (there is at most one outstanding call originally issued by a process).

Basic data structures

The data structure defined in this part will be available to all components of the system.

Universe of the values

Here, the universe values are of different kinds, as memory values, locations, exceptions, procedure names, which are given by separate specifications as follows.

VALUE = *memory values*
requirement
sort value
cn Init_Value: value *value initially contained in the locations*
end

LOCATION = *locations*
requirement sort location **end**

EXCEPTION = *exceptional values*
enum exception: BadArg MemFailure BadCall RPCFailure **end**

[**enum** srt: Id1 ... Idn **end** is a METAL shortcut denoting the specification of a data structure with just the sort srt, whose elements are exactly Id1, ..., Idn.]

PROC = *procedure names*
design
use NAT
sort proc
cn Write, Read: proc
given a procedure name returns the number of its arguments
op ArgNum: proc -> nat
ax ArgNum(Read) = 1
ax ArgNum(Write) = 2
end

UNIVERSE =
universe of the values either arguments or results of the procedures of MC *or of its components*
design
use VALUE, LOCATION, EXCEPTION, PROC
sort universe

a memory value, a location, an exception and a procedure name are universe values
op V: value -> universe
op L: location -> universe
op E: exception -> universe
op P: proc -> universe

cn Write_End: universe   *value returned by a successful Write call*

sort list   *lists of values*

cn Empty_List: list   *empty list*
op _ _: universe list -> list   *adds a value to a list*

op Length: list -> nat   *given a list returns its length*
ax Length(Empty_List) = 0
ax Length(u ls) = Length(ls) + 1

a list of values is a value
op LS: list -> universe

check whether a universe element is a list of a given length
pr Is_List: universe nat
ax if n = Length(ls) then Is_List(LS(ls), n)

check whether a universe element is not RPCFailure
pr Is_Not_RPCFailure: universe
ax Is_Not_RPCFailure(L(l))
ax Is_Not_RPCFailure(V(v))
ax Is_Not_RPCFailure(P(p))
ax Is_Not_RPCFailure(LS(ls))
ax Is_Not_RPCFailure(E(BadArg))
ax Is_Not_RPCFailure(E(MemFailure))
ax Is_Not_RPCFailure(E(BadCall))
end

Process identifier

PID =
identifiers of the processes (in some other components) originally issuing the calls
requirement sort pid **end**

Calls

CALL = *calls of the* MC *procedures*
design
use UNIVERSE
sort call
op Write: universe universe -> call
op Read: universe -> call
end

Remote calls

REMOTE_CALL = *remote calls received by* RPC
design
use UNIVERSE

sort rcall
the remote calls are made by two universe elements (the procedure name and the list of the arguments)
op $< \_ ; \_ >$: universe universe -> rcall

pr Remote_Correct: rcall *checks if a remote call is correct*
if u is a list whose length is the number of arguments of p, then
a remote call consisting of p and u is correct
ax if Is_List(u, ArgNum(p)) then Remote_Correct(P(p),u)
end

Components of MC

MC is a concurrent system with three components, which in turn are other dynamic systems: CLERK, RMC and RPC.

The reliable memory component RMC

MC' is a specification of the memory component defined as in Sect. 4, except that now the basic data structures universe of values and calls are those specified in this development step; clearly the axioms have to be slightly changed to use the new data; e.g. axiom 1 becomes

ax if not Correct(c) and mc – RECEIVE(c,pi) –> mc′ then
  mc′ in any case $<$ No_Concern(y,pi) $>$ until
    ($<$ RETURN(E(BadArg),pi) $>$ or $<$ RETURN(E(MemFailure),pi) $>$)

RMC =
requirement
use rename sort mc to rmc in MC'

** *RM1* **
RMC *cannot return MemFailure*
ax not rmc – RETURN(E(MemFailure),pi) –> rmc′

** *RM2* **
in any case RMC *will eventually*
ax rmc in any case eventually
reach a state where may receive any call
[(forall c, pi: exists x′: x – RECEIVE(c,pi) –> x′)
end

Axiom RM1 requires the reliability; while axiom RM2, an acceptance vitality property, is needed to use RMC to build the implementation of MC; indeed it avoids that a process which will be never able to receive a call may be chosen to realize RMC.

The remote procedure caller RPC Also in this case the structure of the specification is graphically reported in Fig. 3; there the square boxes enclosed in the rounded box corresponding to RPC represent the basic static structures of MC used by RPC.

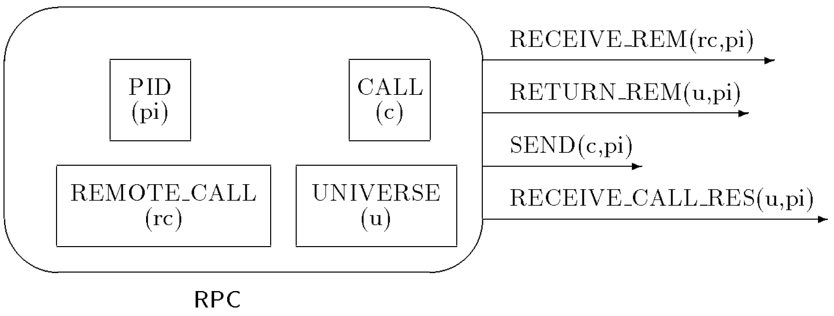

Fig. 3. The specification of RPC

The actual parameter
The specification APAR, used to instantiate the parametric specification RPC_P to get the specification of RPC, defines the data used by such process, precisely the universes of values, the process identifiers, the calls and the remote calls.

APAR =
design
use UNIVERSE, PID, CALL, REMOTE_CALL
transforms a remote call into the corresponding call
op Call: rcall -> call
ax Call(< P(Read) ; LS(u Empty_List) >) = Read(u)
ax Call(< P(Write) ; LS(u u′ Empty_List) >) = Write(u,u′)
end

RPC = RPC_P(APAR)   *the* RPC *used for implementing* MC

The above specification is correct, since APAR is a correct parameter for RPC_P, i.e. its signature includes that of PAR and the axioms of PAR hold in APAR.

The clerk CLERK Also in this case the structure of the specification is graphically reported in Fig. 4.

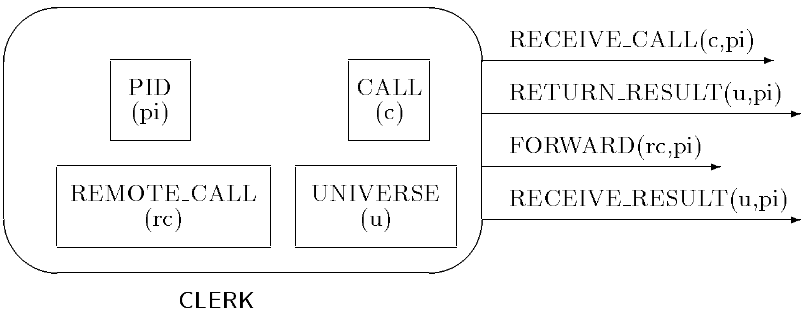

Fig. 4. The specification of CLERK

Interactions We precisely know the interactions of CLERK with its external world, and so we use a design specification for them.

CLERK_INTERACT =
design
use PID, CALL, REMOTE_CALL, UNIVERSE
sort lab_clerk

to receive (from outside MC*) a call and an identifier (of the process in some other component that has originally issued the call)*
op RECEIVE_CALL: call pid -> lab_clerk

to return (outside MC*) a result and an identifier (of the process in some other component that finally will get the result)*
op RETURN_RESULT: universe pid -> lab_clerk

to forward (to RPC) a remote call and an identifier (of the process in some other component that has originally issued the call)
op FORWARD: rcall pid -> lab_clerk

to receive (from RPC) a result (of a remote call) and an identifier (of the process in some other component that finally will get the result)
op RECEIVE_RESULT: universe pid -> lab_clerk
end

States

CLERK_STATE = **requirement dsort** clerk: _ – _ –> _ **end**

Activity

CLERK =
requirement
use CLERK_INTERACT, CLERK_STATE

auxiliary operation, given a call returns the corresponding remote call
op Remote: call -> rcall
ax Remote(Read(u)) = < P(Read) ; LS(u Empty_List) >
ax Remote(Write(u,u′)) = < P(Write) ; LS(u u′ Empty_List) >

auxiliary predicate
given a clerk state cl, a process identifier pi and a call c, checks if c is the last call received with pi in cl (notice that it is also the only outstanding one concerning pi)
pr Is_Last_Call: clerk call pid
ax Is_Last_Call(cl,c,pi) iff
  in any case cl
  cl in any case
    has not received another call and pi since
    < not exists c′: y = RECEIVE_CALL(c′,pi) > since
      has received c and pi
      < RECEIVE_CALL(c,pi) >

*** C1 ***
if CLERK *receives a call c and pi, then*
ax if cl – RECEIVE_CALL(c,pi) –> cl′ then
  in any case it will eventually
  cl′ in any case eventually
    forward the remote call corresponding to c and pi
    < FORWARD(Remote(c),pi) >

** C2 *
if u is not RPCFailure and CLERK *receives u and pi, then*
ax if Is_Not_RPCFailure(u) and cl – RECEIVE_RESULT(u,pi) –> cl′ then
in any case it will eventually return result u and pi
cl′ in any case eventually < RETURN_RESULT(u,pi) >

** C3 **
if CLERK *receives RPCFailure and pi, then*
ax if cl – RECEIVE_RESULT(RPCFailure,pi) –> cl′ then
in any case it will eventually
cl′ in any case eventually
either forward the remote call corresponding to
(< exists c: y = FORWARD(Remote(c),pi) and
the last call received with pi and pi
Is_Last_Call(cl′,c,pi) >
or return result MemFailure and pi
or < RETURN_RESULT(E(MemFailure),pi) >)

** C4 **
if CLERK *forwards a remote call rc and pi, then*
ax if cl – FORWARD(rc,pi) –> cl′ then
there exists a call c s.t. rc is its corresponding remote call and
exists c: rc = Remote(c) and
in any case it
cl′ in any case
has not forwarded any remote call with pi since
(< not exists rc′: FORWARD(rc′,pi) > since
either has received c and pi
(< RECEIVE_CALL(c,pi) >
or has received RPCFailure and pi and
or (< RECEIVE_RESULT(RPCFailure,pi) > and
c is last call received with pi
Is_Last_Call(cl′,c,pi))))

** C5 **
if CLERK *returns a result u different from MemFailure and pi, then*
ax if cl – RETURN_RESULT(u,pi) –> cl′ and not(u = MemFailure) then
in any case it
cl′ in any case
has not returned any result with pi since
(< not exists u′: y = RETURN_RESULT(u′,pi) > since
it has received a call and pi and
< exists c: y = RECEIVE_CALL(c,pi) >) and
has not received any other result and pi since
(< not exists u′: y = RECEIVE_RESULT(u′,pi) > since
it has received u and pi
< RECEIVE_RESULT(u,pi) >)

*** C6 ***
if CLERK *returns the result MemFailure and pi, then*
ax if cl – RETURN_RESULT(MemFailure,pi) –> cl′ then
in any case it
cl′ in any case
has not returned any result with pi since
(< not exists u′: y = RETURN_RESULT(u′,pi) > since
it has received a call with pi and
< exists c: y = RECEIVE_CALL(c,pi) >) and
has not received any other result with pi since
(< not exists u′: y = RECEIVE_RESULT(u′,pi) > since
it has received RPCFailure and pi
< RECEIVE_RESULT(RPCFailure,pi) >)

*** C7 ***
if CLERK *receives a call c and pi, then*
ax if cl – RECEIVE_CALL(c,pi) –> cl′ then
in any case it will eventually return a result
cl′ in any case eventually < exists u: y = RETURN_RESULT(u,pi) >

*** C8 ***
in any case CLERK *will eventually*
ax cl in any case eventually
reach a state where may receive any result
[forall u, pi: exists cl′: x – RECEIVE_RESULT(u,pi) –> cl′]
end

Interactions (of MC)

In this development step we precisely fix the interactions of MC with its external world, and so we use a design specification for them; notice that now MC has one interaction not present in the requirement specification (INT).

MC_INTERACT =
design
use CALL, PID
sort lab_mc
to receive a call and a process identifier (of the process in some other component that has originally issued the call)
op RECEIVE: call pid -> lab_mc
to return a result and a process identifier (of the process in some other component that finally will get the result)
op RETURN: universe pid -> lab_mc
to perform an internal action
cn INT: lab_mc
end

States (of MC)

In this development step we also precisely fix the intermediate states of MC; they are just given by the states of its three components, and so we use a design specification for them.

MC_STATE =
design
use RPC, CLERK, RMC
dsort mc: _ – _ –> _
op _ | _ | _: rmc rpc clerk -> mc
end

Activity (of MC)

The requirements on the activity of MC in this development step are different from those on its components; indeed here we have to express properties on the activity of a concurrent dynamic system, relating the fact that a component perform an action with the fact that another component performs another action (e.g. saying that if CLERK forwards a remote call, such call must be received by RPC). Thus we need a kind of "distributed" temporal logic, with combinators for saying e.g. that a component of a concurrent system will eventually perform some action. Since we know the distributed structure of MC (it has three components) we can use the logic presented in Sect. 2.3; indeed the following path formula

exists rmc, rpc, cl, rmc′, rpc′, cl′:

[x = rmc | rpc | cl] and next [x = rmc′ | rpc′ | cl′ and rpc – lp –> rpc′]

holds on a path of MC whenever in the first transition of such path the component RPC has performed a transition labelled by lp; in the following such formula will be simply written as < RPC: lp >. Clearly, we have to assume that the formula

if rpc – lp –> rpc′ and rpc – lp' –> rpc′ then lp = lp'

is satisfied by the RPC specification; but this requirement is not problematic, because any specification of a dynamic system without this property may be transformed into another one with such property and behaving equivalently.

Analogous abbreviations can be defined for the RMC and CLERK components, written < RMC: lm > and < CLERK: lc > respectively.

[Below "some time" is the METAL syntax for the combinator of the temporal logic $\Diamond \mathbf{P}$, introduced in Sect. 2.3.]

MC =
requirement
use MC_STATE, MC_INTERACT

The activity of MC *is fully determined by those of its components*

** 1’ **
in any case if MC *performs an action, then*
ax mc in any case if < l > then
    at least one of the components performs an action
    (exists lc: < CLERK: lc >) or
    (exists lp: < RPC: lp >) or
    (exists lm: < RMC: lm >)

MC *cannot stop its components forever*

** 2’ **
if CLERK *can perform some action, then*
ax if exists cl′, lc: cl – lc –> cl′ then
    in any case it will eventually perform some action
    cl | rpc | rmc in any case eventually exists lc″: < CLERK: lc″ >

** 3’ **
if RPC *can perform some action, then*
ax if exists rpc′, lp: rpc – lp –> rpc′ then
    in any case it will eventually perform some action
    cl | rpc | rmc in any case eventually exists lp″: < RPC: lp″ >

** 4’ **
if RMC *can perform some action, then*
ax if exists rmc′, lm: rmc – lm –> rmc′ then
    in any case it will eventually perform some action
    cl | rpc | rmc in any case eventually exists lm″: < RMC: lm″ >

CLERK *takes care of the interactions of* MC *with the external world*

** 5’ **
in any case
ax mc in any case
    if MC *receives a call and pi, then* CLERK *receives them*
    if < RECEIVE(c,pi) > then < CLERK: RECEIVE_CALL(c,pi) >

** 6’ **
in any case
ax mc in any case
    if CLERK *receives a call and pi, then* MC *receives them*
    if < CLERK: RECEIVE_CALL(c,pi) > then < RECEIVE(c,pi) >

*** 7' ***

in any case

ax mc in any case

if MC *returns a result and pi, then* CLERK *returns them*

if < RETURN(u,pi) > then < CLERK: RETURN_RESULT(u,pi) >

*** 8' ***

in any case

ax mc in any case

if CLERK *returns a result and pi, then* MC *returns them*

if < CLERK: RETURN_RESULT(u,pi) > then < RETURN(u,pi) >

*** 9' ***

in any case

ax mc in any case

if MC *performs an internal action, then*

if < INT > then

CLERK *neither returns a result nor receives a call*

((not exists u, pi: < CLERK: RETURN_RESULT(u,pi) >) and

(not exists c, pi: < CLERK: RECEIVE_CALL(c,pi) >))

Cooperation between CLERK *and* RPC

*** 10' ***

in any case

ax mc in any case

if CLERK *forwards a remote call and pi, then*

if < CLERK: FORWARD(rc,pi) > then

RPC *will eventually receive them*

eventually < RPC: RECEIVE_REM(rc,pi) >

*** 11' ***

in any case

ax mc in any case

if RPC *receives a remote call and pi, then*

if < RPC: RECEIVE_REM(rc,pi) > then

some time CLERK *has forwarded them*

some time < CLERK: FORWARD(rc,pi) >

*** 12' ***

in any case

ax mc in any case

if RPC *returns a result and pi, then*

if < RPC: RETURN_REM(u,pi) > then

CLERK *will eventually receive them*

eventually < CLERK: RECEIVE_RESULT(u,pi) >

*** 13' ***
in any case
ax mc in any case
if CLERK *receives a result and pi, then*
if $<$ CLERK: RECEIVE_RESULT(u,pi) $>$ then
some time RPC *has returned them*
some time $<$ RPC: RETURN_REM(u,pi) $>$

Cooperation between RPC *and* RMC

*** 14' ***
in any case
ax mc in any case
if RPC *sends a call and pi, then*
if $<$ RPC: SEND(c,pi) $>$ then
RMC *will eventually receive them*
eventually $<$ RMC: RECEIVE(c,pi) $>$

*** 15' ***
in any case
ax mc in any case
if RMC *receives a call and pi, then*
if $<$ RMC: RECEIVE(c,pi) $>$ then
some time RPC *has sent them*
some time $<$ RPC: SEND(c,pi) $>$

*** 16' ***
in any case
ax mc in any case
if RMC *returns a result and pi, then*
if $<$ RMC: RETURN(u,pi) $>$ then
RPC *will eventually receive them*
eventually $<$ RPC: RECEIVE_CALL_RES(u,pi) $>$

*** 17' ***
in any case
ax mc in any case
if RPC *receives a result and pi, then*
if $<$ RPC: RECEIVE_CALL_RES(u,pi) $>$ then
some time RMC *has sent them*
some time $<$ RMC: SEND(c,pi) $>$
end

Notice that property 10' implicitly requires also that RPC will eventually have the capability to receive a remote call; but that is already ensured by the RPC properties (axiom R7), so the specification of MC is "conservative" w.r.t. that of

RPC. Similarly, the properties 12', 14' and 16' require that some component must have some action capabilities, but such capabilities are already ensured by axioms C8, RM2 and R8.

Correctness justification

The check of the correctness of the development step follows the specification structure, so first we consider the basic data structures, then states, interactions and finally the properties on the activity. To improve the readability in the following we will add the suffix either 1 or 2 to the names of the specifications to distinguish those defined in the requirement phase (Sect. 4) and those in this development step.

Basic data Structures

The signatures of UNIVERSE1 and of UNIVERSE2 are different; thus we have to add to UNIVERSE2

cn Init_Value: universe
cn BadArg, MemFailure: universe
pr Is_MemVal, Is_Loc: universe

defined by the axioms

ax Init_Value = V(Init_Value)
ax BadArg = E(BadArg)
ax MemFailure = E(MemFailure)
ax Is_MemVal(V(v))
ax Is_Loc(L(lc))

and to hide all symbols not present in UNIVERSE1.

This modification of UNIVERSE2 can be expressed as a function from specifications into specifications, see at the end of Sect. 2.4. Then it is very easy to verify that all axioms of UNIVERSE1 hold in the modified version of UNIVERSE2.

Similarly we can see that CALL2 is a correct implementation of CALL1. In this case we have to add to CALL2 the predicate

pr Correct: call

defined by the axioms

ax Correct(Write(L(lc),V(v)))
ax Correct(Read(L(lc))).

PID2 coincides with PID1.

States

To see that MC_STATE2 is a correct implementation of MC_STATE1 we have to add to MC_STATE2 the operation

op Cont: mc universe -> universe

defined by

ax Cont(cl | rpc | rmc) = Cont(rmc)

and to hide the operation _ | _ | _.

Interactions

To see that MC_INTER2 is a correct implementation of MC_INTER1 we have to add to MC_INTER2 the constant **cn** INT: lab_mc.

Activity

For lack of room we only report the proof that axiom 1 holds in MC2; those for the other axioms are analogous.

By axiom 5', if MC2 performs RECEIVE(c,pi) with c non correct, then CLERK performs RECEIVE_CALL(c,pi).

By axioms C1, 2' and 1', CLERK will eventually perform FORWARD(Remote(c),pi).

By axiom 10', RPC will eventually perform RECEIVE_REM(Remote(c),pi).

Since Remote(c) is correct as remote call, by axioms 3', 2' and R1 we have two possible cases:

1. RPC will eventually perform SEND(c,pi).
   By axiom 14', RMC will eventually perform RECEIVE(c,pi).
   Since c is not correct, by axiom 1 RMC will eventually perform RETURN(u,pi), with u either equal to E(BadArg) or to E(MemFailure).
   By axiom 16', RPC will eventually perform RECEIVE_CALL_RES(u,pi).
   By axioms R3 and 3', we have two possible cases:
   (a) RPC will eventually perform RETURN_REM(u,pi).
       By axiom 12', CLERK will eventually perform RECEIVE_RESULT(u,pi).
       By axiom C2 and 2', CLERK will eventually perform RETURN_RESULT(u,pi).
       By axiom 8', MC2 will perform RETURN(u,pi).
       OK
   (b) RPC will eventually perform RETURN_REM(E(RPCFailure),pi)
       By axiom 12', CLERK will eventually perform RECEIVE_RESULT(E(RPCFailure),pi).
       By axiom C3 we have two cases:
       i. CLERK will eventually perform RETURN_RESULT(E(MemFailure),pi).
          By axiom 8', MC2 will perform RETURN(E(MemFailure),pi).
          OK
       ii. CLERK will eventually perform FORWARD(Remote(c),pi).
          Then the proof goes on as from the beginning; axiom C7 prevents the case in which forever the second alternative is taken; thus a result will be returned eventually.
          OK
2. RPC will eventually perform RETURN_REM(E(RPCFailure),pi).
   As in 1b.
   OK

We have still to prove that before returning the right result MC2 does not perform any other interaction concerning pid; that follows from the properties of the various components prohibiting "unsolicited reactions" as R4, R5, C4 and C5 and by property 1' ensuring that all transitions of MC2 are due to transitions of its components.

7 Discussing our Solution

Here we try to present the main features of our solution of RPC-Memory Specification Problem, highlighting the positive and negative aspects.

Structured/modular specification The specifications presented in this paper are structured and modular, moreover their structures correspond to the structures of the specified systems; that allows also the possibility of reusing specification parts (see, e.g. the parameterized RPC component).

Uniform treatment of specifications at different levels The specifications of the system at different levels of abstraction, in our terminology requirement and design specifications, have the same structure and the specification language treats similarly the common parts (the only difference is in the form of the axioms and in the intended semantics of the specifications).

Adequacy to the RPC-Memory Specification Problem One of the relevant feature of our solution is that it takes into account all parts and aspects of the RPC-Memory Specification Problem (Sect. 1, 2, and 3 of [6]); so we have formalized *exactly* RPC-Memory Specification Problem, except for the "shadow spots" and without having to extend/modify the method. For example:

- MC is seen as an "open" system interacting with its external environment;
- the non-concurrent aspects, as those about the data used by the components, have been considered;
- reusable (parameterized) parts, as the RPC, may be specified including the properties on the parameters;
- concurrent/distributed systems, as the first refinement of [6], are explicitly handled.

In [6] further development steps were considered (see Sect. 4); but there (real) time aspects were involved and so we have not considered them. Indeed, in SMoLCS there are no special features for handling the (real) time aspects, so the specification about this part is not standard. It can be done under the assumption that the duration of all atomic actions of all components is given using a common discrete time unit. If this assumption is sensible, then SMoLCS specifications may handle the timed features (see, e.g. [12], which presents the specification of a controller of a system performing various checks each 5, 30, ... minutes).

Instead, also in order to show how our method works at the later stages of the development, in [5] we have worked out, making some particular choices, two further development steps.

Second development step We start to design the system specified in Sect. 6 by assuming that its components cooperate by synchronous message exchange and that perform their activities in a free independent way (i.e. there are no overall constraints on them).

Third and last development step We fully design the three components of the system specified above, CLERK, RPC and RMC, and so we get a complete design, i.e. a complete definition, of the MC that we have worked out.

Prototyping of the developed system The MC determined above has been tested by using the prototyping tools, see [1].

No overspecification Our specification expresses only what is contained in the RPC-Memory Specification Problem text in [6]; no need to embed in the specification other features, like particular kind of message exchange between the components of a system and particular kinds of scheduling of the activity of such components. On the contrary, the features that one usually forgets to mention when describing a system, since they seem obvious but are not guaranteed by default, are explicitly expressed by our specification (e.g. the point that the memory cells may or may not fail). Moreover each feature is directly expressed in the specification, and not by means of either a coding or a low level implementation.

Readability We think that our solution is rather readable due to the two-rail approach (informal and formal specifications), to the structuring of the specification, and to the syntactic richness and friendliness of the metalanguage METAL (e.g. distributed infix syntax, absence of "esoteric" symbols).

Long specification text Our specification, to have the advantages listed in the previous points, is long, and in some sense also complex (that is due mainly to the presence of some overhead, as the typed nature of the metalanguage, which requires an explicit declaration of all used symbols). Such unpleasantness may be overcome by using interactive editors/browers, and using the specification structure as a basis for giving a hypertext version of the documents produced during the development; so to be able, e.g. to drop the informal comments/formulae just by a click of the mouse.

About "correctness", "validation", ... Our method is still much unfinished and lacking in this respect, since we do not have a standard proof assistant. We have plans for establishing a friendly connection with some general tool for associated proofs, like PVS.

Here we briefly summarize what we have done for what concerns the general problem of correctness/validation using the supports (e.g. software tools) offered by the SMoLCS methodology to help this task.

We have a parser and a type checker for the SMoLCS specification language METAL; and so we have first controlled the static correctness of our specifications detecting some errors concerning a conceptual confusion between "calls" and "remote calls". We have fixed them, by introducing two distinct data structures for calls and for remote calls.

After we have tried to see if our design specification given in PHASE 2 was correct w.r.t. the requirements given in PHASE 1, i.e. to give the "correctness justification part". The methodology does not offer either software tools (as theorem provers and model checkers) or theoretical ones (as sound and complete deductive systems for the used logics and refinement calculi). The proof has to be done by hand and presented using the natural language, as it usually done for proving a theorem of analysis. We have done this proof, which is briefly reported in Sect. 6, detecting some errors.

Our specification of MC (see Sect. 4) allows that the memory component may refuse to receive calls forever; but such kind of component cannot work as part of the implementation. This point has been solved by adding property RM2 to the specification of RMC, see Sect. 6.

Another error found during the proof is the following. The first version of the MC specification required that in case of incorrect calls the exception RPCFailure cannot be returned, while the implementation may return RPCFailure for any call. We have simply fixed it by changing the requirements on MC.

Harder has been to prove the correctness of the further development steps, but in such case we have heavily used the rapid prototyper to analyze the behaviour of the design system. We have immediately found that there were several problems in earlier versions, mainly deadlocks, due to RPC and to CLERK.

Has the use of SMoLCS really worked? That means, have we got any benefits by using SMoLCS to develop MC? Also in this case SMoLCS presents the benefits already found in previous applications:

- it obliges the client to clarify a lot its idea of the system to be developed. In this case all shadow spots are questions for the client; e.g. it is relevant the point about the failures. Furthermore these questions may be discussed with the client using only the informal part of the specification.
- it helps to validate the implementation; we have found several errors in a first version of the implementation, also without tools for automatizing the proof.

Acknowledgement We warmly thank Stefano Ferrua for helping to prove the correctness of the implementation.

References

1. E. Astesiano, F. Morando, and G. Reggio. The SMoLCS Toolset. In Mosses P.D., Nielsen M., and Schwartzbach M.I., editors, *Proc. of TAPSOFT '95*, number 915 in LNCS, pages 810–801. Springer Verlag, Berlin, 1995.
2. E. Astesiano and G. Reggio. SMoLCS-Driven Concurrent Calculi. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proc. TAPSOFT'87, Vol. 1*, number 249 in LNCS, pages 169–201. Springer Verlag, Berlin, 1987.
3. E. Astesiano and G. Reggio. Formally-Driven Friendly Specifications of Concurrent Systems: A Two-Rail Approach. Technical Report DISI–TR–94–20, DISI – Università di Genova, Italy, 1994. Presented at ICSE'17-Workshop on Formal Methods, Seattle April 1995.
4. E. Astesiano and G. Reggio. Algebraic Dynamic Specifications: An Outline. Technical Report DISI–TR–95–08, DISI – Università di Genova, Italy, 1995.
5. E. Astesiano and G. Reggio. A Dynamic Specification of the Complete Development of the RPC-Memory. Technical Report DISI–TR–96–02, DISI – Università di Genova, Italy, 1996.
6. M. Broy and L. Lamport. The RPC-Memory Specification Problem. In this volume.
7. G. Costa and G. Reggio. Abstract Dynamic Data Types: a Temporal Logic Approach. In A. Tarlecki, editor, *Proc. MFCS'91*, number 520 in LNCS, pages 103–112. Springer Verlag, Berlin, 1991.
8. G. Costa and G. Reggio. Specification of Abstract Dynamic DataTypes: A Temporal Logic Approach. *T.C.S.*, 1996. To appear.
9. Z. Manna and A. Pnueli. The Anchored Version of the Temporal Framework. In J.W. de Bakker, W.-P. de Roever, and G. Rozemberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 354 in LNCS. Springer Verlag, Berlin, 1989.

10. F. Parodi and G. Reggio. METAL: a Metalanguage for SMoLCS. Technical Report DISI–TR–94–13, DISI – Università di Genova, Italy, 1994.
11. G. Reggio, D. Bertello, and A. Morgavi. The Reference Manual for the SMoLCS Methodology. Technical Report DISI–TR–94–12, DISI – Università di Genova, Italy, 1994.
12. G. Reggio and E. Crivelli. Specification of a Hydroelectric Power Station: Revised Tool-Checked Version. Technical Report DISI–TR–94–17, DISI – Università di Genova, Italy, 1994.
13. M. Wirsing. Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoret. Comput. Sci.*, volume B, pages 675–788. Elsevier, 1990.