Applying a Temporal Logic to the RPC-Memory Specification Problem

Reinhard Gotzhein

Department of Computer Science, University of Kaiserslautern
PO 3049, D-67653 Kaiserslautern, Germany

Abstract. The unreliable and the reliable memory component of the RPC-Memory Specification Problem are specified and verified using a formalism that is based on a temporal logic. In that formalism, a system (open and/or distributed) consists of the system architecture and the system behaviour. The architecture is specified by defining the sets of its agents (= active components) and interaction points (= conceptual locations of interactions), and by associating with each agent a set of interaction points. The behaviour is built from external actions and defined by the logical conjunction of all component behaviours. A component behaviour is specified by the conjunction of logical safety and liveness properties. For the memory component, a many-sorted first-order branching time temporal logic with operators to refer to the future, the past, actions, the number of actions, and intervals constructed in the (linear) past are used. It is proven that the reliable memory is a correct implementation of the unreliable memory. The proof obligation takes both the system architecture and the system behaviour into account. Finally, it is discussed whether architectural requirements should be formally expressed.

1 Introduction

The RPC-Memory Specification Problem calls for the specification and verification of a series of components that can be embedded into a larger context that is not known in advance. From the problem description, we infer that it is important to specify how this embedding may be done, and what behaviour the components of the context may perceive when the specified components are stimulated. To formally specify these aspects, we have chosen a formalism based on temporal logic that is able to specify both architectural and behavioural aspects. Furthermore, the formalism suggests a notion of system refinement that involves both architecture and behaviour. This is different from other techniques in this volume which do not give a semantics to structural aspects of a specification and thus to the architecture of the specified system. We argue that specifying the essential system architecture should be among the first development steps, and be as rigorous and formal as the specification of the system behaviour.

In this paper, solutions for Problems 1a, 1b, and 1c are presented in detail. The specifications are formal, the proof is straightforward and involves simple temporal reasoning. Problem 3 is briefly addressed. Problems 4 and 5 are not treated because they include bounded response properties which at the moment are not expressable in our formalism. Although it is technically feasible to solve problems 2 and 3, we have not developed solutions. The reason is that the refinement as suggested in the problem statement introduces a system structure and corresponding behaviour which in our opinion is not a requirement, but one of several possible design decisions. This step is not well supported by our formalism, which should be used for the specification of requirements only. We will explain this in more detail in Sections 3.3 and 4.

The structure of the paper is the following. Section 2 gives a brief introduction to the formalism and its underlying concepts. In Section 3, this formalism is applied to some parts of the RPC-Memory Specification Problem. In Section 4, we discuss whether architectural requirements should be expressed formally. This aspect is not treated in other solutions listed in this volume, and has been a point of controversy during and after the Dagstuhl workshop. Section 5 draws a conclusion.

2 The formalism

In this section, the formalism applied to the RPC-Memory Specification Problem will be explained. It can be classified as a property-oriented formal description technique, suitable for the specification of requirements for open distributed systems. The view taken throughout this paper and questioned in Section 4 is that these systems consists of system behaviour *and* system architecture. Elementary concepts are used to define these system aspects. For further details and examples, including the modeling, specification and verification of a communication service, see [7].

2.1 Elementary concepts

We start with the informal introduction of a very small number of elementary concepts. Due to their elementary nature, these concepts cannot be formally defined. All that can be stated at this point is that these concepts are disjoint. Elementary concepts form the basis for the definition of several derived concepts.

Definition 1 (elementary concepts).
- An *agent* $ag \in Ag$ is a component performing actions.
- An *interaction point* $ip \in Ip$ is a conceptual location where actions may occur[1].
- An *action*[2] $a \in Act$ is something that happens.
- Agent, interaction point, and action are disjoint concepts, i.e. $Ag \cap Ip = Ag \cap Act = Ip \cap Act = \{\}$.

[1] A specialization of the interaction point concept called "procedure interface" is used in [1].

[2] sometimes called *action occurrence*

An action is performed by an agent or a set of agents, it may be internal or external. External actions occur at some interaction point or a set of interaction points. An agent thus is the carrier of actions, it can be characterized by its behaviour. This behaviour (a notion still to be defined) consists of actions local to the agent, i.e. internal and external actions of the agent. For instance, the share an agent has in an interaction is considered external. Actions may also be non-local, such as interactions or transactions as a whole. Non-local actions may be performed by a *set* of agents and may occur at a *set* of interaction points. Interactions and transactions may also be considered as high-level actions, i.e., actions that can be decomposed into smaller units. Depending on what kind of action is taken as atomic on a given level of abstraction, the behaviour of a system can be characterized in different ways.

2.2 System architecture

With the elementary concepts *agent* and *interaction point*, more complex structures termed *system architectures* can be composed:

Definition 2 (system architecture). A *system architecture* $Arch$ is a structure $\langle Ag, Ip, ArchF \rangle$, where
- Ag is a non-empty set of agents,
- Ip is a set of interaction points, and
- $ArchF : Ag \to 2^{Ip}$ is a total function called *architecture function* associating with each agent a set of interaction points.

A set $Ag^* \subseteq Ag$ of agents has one or more interaction points in common if and only if $\bigcap_{ag \in Ag^*} ArchF(ag) \neq \{\}$. We require as a rule of composition that a common interaction point is introduced explicitly into the architecture whenever a group of agents has the capability to interact directly. Depending on the kind of interaction, two or more agents may in general be involved in interactions. Whether such interactions will actually take place also depends on the behaviour.

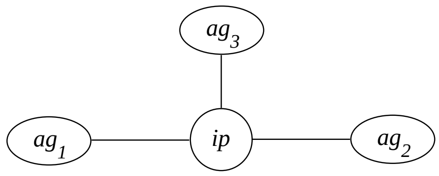


Fig. 1. Graphical representation of an architecture

Figure 1 shows a graphical representation of an architecture $Arch = \langle \{ag_1, ag_2, ag_3\}, \{ip\}, ArchF(ag_1) = ArchF(ag_2) = ArchF(ag_3) = \{ip\} \rangle$ consisting of three agents ag_1, ag_2, and ag_3 that have a common interaction point ip. From this architecture, we can infer that ag_1, ag_2, and ag_3 have the capability to interact, however, we can not yet say whether they will actually do so. This can only be derived from the behaviour of the agents and the interaction point.

With the definition of the system architecture, important structural information about the system to be specified is determined. Since the architecture is part of the requirements, it will have to be respected in subsequent refinement steps. Thus, system architecture will have a semantics, which is taken into account in the correctness notion explained in Section 2.5. Different from other approaches, architecture is therefore not just a means to structure a specification. It will become clear in the following that a system specification can have additional structure without a semantics, which therefore needs not be respected in subsequent refinements.

2.3 System behaviour

With the elementary concept *action*, more complex structures termed *behaviours* can be composed. A behaviour is specified as a conjunction of logical formulas, each expressing a restriction on the behaviour of system components (agents and interaction points). The behaviour refers to external actions, which are associated with system components and therefore contain some reference to the system architecture. In the following definition, we assume that restrictions are syntactically local in the sense that they each refer to aspects of a single system component only: a property referring to local aspects of an agent (interaction point) contains only references to that agent (interaction point) or its attached interaction points (agents)[3]. As a consequence, we can compose system behaviour from component behaviour, which supports modular system development and verification. Additionally, non-local properties can be specified and proven from the composition of component behaviour.

Definition 3 (system behaviour). Let $Arch = \langle Ag, Ip, ArchF \rangle$ be a system architecture, $Behav_{ag}$ be the behavioural specification of agents $ag \in Ag$, and $Behav_{ip}$ be the behaviour of interaction points $ip \in Ip$. The *system behaviour* $Behav$ is given as the conjunction of component behaviour, i.e.:

$$Behav =_{Df} \bigwedge_{ag \in Ag} Behav_{ag} \wedge \bigwedge_{ip \in Ip} Behav_{ip}$$

When system components are specified separately and composed using logical conjunction, the question arises whether this yields a description of the system behaviour. The crucial idea here is the explicit introduction of interaction points as connection links between agents. By this elementary architectural concept,

[3] We do not use an explicit assumption – guarantee format. In general, assumptions about the environment will have to be made in order to implement syntactically local restrictions.

sets of local actions forming interactions can be related. How they are related depends on the system architecture and the behaviour of interaction points.

To specify component behaviour, a variety of logics can be used. Since behaviour involves a notion of time, a temporal logic is a possible candidate. Several kinds of temporal logics have been proposed, including linear time and branching time temporal logic, temporal logic based on propositional logic and first-order logic, and for each of these alternatives, a variety of temporal operators to refer to the future, the past, actions, and intervals [6]. These logics differ in expressiveness, availability and completeness of axiomatic bases, and decidability. Consequently, the logic for the specification of component behaviour should be selected carefully with respect to the given problem.

In [7], we have introduced a number of building blocks, each consisting of formation rules, semantics, and an axiomatic basis, which together form a modular temporal logic:

- At_P atomic formulas of propositional logic,
- At_F atomic formulas of first-order logic,
- Pr propositional operators,
- Qu quantifiers,
- F_L linear time temporal operators to refer to the future,
- F_B branching time temporal operators to refer to the future,
- P_L temporal operators to refer to the past,
- A_P action operator,
- A_F operator for the number of actions,
- I operators for interval construction.

For a given problem, a subset of these building blocks can be selected and combined, i.e., the formation rules, semantics, and axiomatic bases are joined together, in order to customize an appropriate temporal logic. A few obvious restrictions apply to the combination of building blocks:

- exactly one of the building blocks At_P and At_F must be selected;
- selection of Pr is mandatory;
- only one of the building blocks F_L and F_B may be selected;
- A_F can not be combined with At_P;
- if I is chosen, A_P must also be selected.
- A_F can not be combined with At_P;
- if I is chosen, A_P must also be selected.

The combination of several groups of temporal operators yields a so-called *mixed* temporal logic. While the formation rules and semantics of a mixed logic obtained by joining together the formation rules and semantics of the selected building blocks are complete, the axiomatic basis needs supplement.

When customizing a temporal logic, the following heuristics are helpful to make an adequate choice:

- select At_F, if architectural aspects have to be expressed in formulas;
- select Qu for quantification over infinite value domains;
- select F_L to express invariance and liveness for reliable systems;
- select F_B to express invariance and liveness for unreliable systems;
- select P_L to avoid history variables;
- select A_P to express actions;
- select A_F to count actions without introducing history variables;
- select I to express contexts in the past and thus to avoid history variables.

For the RPC-Memory Specification Problem, we customize a many-sorted first-order branching time temporal logic with operators to refer to the future, the past, actions, the number of actions, and intervals, i.e. building blocks At_F, Pr, Qu, F_B, P_L, A_P, A_F, and I are selected. Note that we use a branching time logic that will allow us to specify potential behaviour, which is not possible in the linear time version. This way, we can express additional liveness requirements in the case of unreliable systems. We will now summarize the customized logic.

In many-sorted first-order temporal logic, atomic formulas are formed starting from the following sets of symbols:

- a denumerable list S of symbols called *sorts*, including the natural numbers $\mathbb{N}_0$.
- a denumerable list V of symbols called *individual variables*; each $x \in V$ is attached to a sort $s \in S$, which is expressed by writing x^s;
- for each integer $n \geq 0$, a denumerable set $\mathcal{F}^{(n)}$ of *n-ary function symbols*; each $f \in \mathcal{F}^{(n)}$ is associated with sorts $s, s_1, ..., s_n \in S$ (written "$f_{s,s_1,...,s_n}$");
- for each integer $n \geq 0$, a denumerable set $\mathcal{R}^{(n)}$ of *n-ary relation symbols*; each $r \in \mathcal{R}^{(n)}$ is associated with sorts $s_1, ..., s_n \in S$ (written "$r_{s_1,...,s_n}$").

With these preparations, the syntax of formulas can be defined as follows:

i) For all $x^s \in V$: x^s is a term of sort s.
ii) For all $n \geq 0$, $f_{s,s_1,...,s_n} \in \mathcal{F}^{(n)}$, and terms $t_1, ..., t_n$ of sorts $s_1, ..., s_n$: $f_{s,s_1,...,s_n}\ (t_1, ..., t_n)$ is a term of sort s.
iii) For all $n \geq 0$, $r_{s_1,...,s_n} \in \mathcal{R}^{(n)}$, and terms $t_1, ..., t_n$ of sorts $s_1, ..., s_n$: $r_{s_1,...,s_n}$ $(t_1, ..., t_n)$ is a formula.
iv) Let φ be a formula, x^s be an individual variable, then $\exists x^s.\varphi$ and $\forall x^s.\varphi$ are formulas.
v) Let φ be a formula, then $\neg\varphi$ is a formula.
vi) Let φ_1, φ_2 be formulas, then $(\varphi_1 \wedge \varphi_2), (\varphi_1 \vee \varphi_2), (\varphi_1 \supset \varphi_2), (\varphi_1 \equiv \varphi_2)$ are formulas.
vii) Let φ be a formula, then $AG\varphi, AF\varphi$ and $EF\varphi$ are formulas.
viii) Let φ be a formula, then $\blacksquare\, \varphi$ and $\blacklozenge\, \varphi$ are formulas.
ix) Let φ be a formula, then $[\varphi]$ (read "action of type φ") is a formula.
x) Let φ be a formula, then $\#[\varphi]$ (read "number of actions of type φ") is a term of sort $\mathbb{N}_0$.
xi) Let φ be a formula, α be an action term, then $[\varphi]$, $\bullet\alpha$ are action terms.

xii) Let α be an action term, β be an interval term, then α, (β) are interval expressions.

xiii) Let δ, δ_1, δ_2 be interval expressions, then $\delta \Rightarrow$, $\delta \Leftarrow$, $\Rightarrow \delta$, $\Leftarrow \delta$, $\delta_1 \Rightarrow \delta_2$, $\delta_1 \Leftarrow \delta_2$, *begin* δ, *end* δ are interval terms.

xiv) Let φ be a formula, β be an interval term, then $\langle\beta\rangle\varphi$ is a formula.

To define the structure of the semantical model and the meaning of formulas, we will use some notational conventions. σ and σ' denote infinite state sequences, σ_i is the $(i+1)$th element of the state sequence σ. $\sigma^{i,j}$ denotes the subsequence $\langle\sigma_i...\sigma_j\rangle$, $\sigma^{0,j} < \sigma'$ expresses that $\sigma^{0,j}$ is a prefix of σ'. π and π' are infinite state trees, $r(\pi)$ is the root state of π. $\sigma \in \pi$ means that σ is a branch of π, so from $\sigma \in \pi$ it follows that $\sigma_0 = r(\pi)$.

The semantics of the temporal operators is defined with respect to a model $\mathcal{M} = \langle E, Q, \Pi, Q_0\rangle$, where

- $E = (E_1, ..., E_n)$ is a family of non-empty sets of objects (containing agents, interaction points, natural numbers, etc.);
- Q is a set of states, where each state is given by a set $\mathcal{F}$ of functions and a set $\mathcal{R}$ of relations on E (the states are not states in the ordinary sense with state variables, but contain only what will be necessary to characterize architecture and behaviour);
- $Q_0 \subseteq Q$ is a non-empty set of initial states;
- Π is a set of infinite state trees with states from Q, root states from Q_0, i.e. $r(\pi) \in Q_0$ for all $\pi \in \Pi$; furthermore, it is required that for all $\pi, \pi' \in \Pi$, $r(\pi) = r(\pi')$ implies $\pi = \pi'$ (or, $\pi \neq \pi'$ implies $r(\pi) \neq r(\pi')$).

Formulas of many-sorted first-order logic are interpreted in a model $\mathcal{M}$ by associating for each state $q \in Q$ sort symbols with sets of objects, function symbols with functions, relation symbols with relations as follows[4]:

- to every sort symbol $s \in S$, a set $D^s \in \{E_1, ..., E_n\}$ is attached; for notational convenience, we will use the same identifiers as sort symbols and to refer to the attached object sets, i.e.: $E_i \in S$ and $D^{E_i} = E_i$;
- for each integer $n \geq 0$: to each n-ary function symbol $f_{s,s_1,...,s_n} \in \mathcal{F}^{(n)}$, a function $f : D^{s_1} \times ... \times D^{s_n} \rightarrow D^s$ is attached;
- for each integer $n \geq 0$: to each n-ary relation symbol $r_{s_1,...,s_n} \in \mathcal{R}^{(n)}$, a relation $r \subseteq D^{s_1} \times ... \times D^{s_n}$ is attached.

The propositional operators $(\neg, \wedge, \vee, \supset, \equiv)$, existential and universal quantification $(\exists, \forall)$ are interpreted as usual. The semantics of the temporal operators and the function $\#[\varphi]$ are defined with respect to a satisfaction relation $\models$, where $\models$ is a relation between $\mathcal{M}$, a tree $\pi \in \Pi$, a sequence $\sigma \in \pi$, positions i, j with $0 \leq i \leq j$, and a formula φ (written "$\mathcal{M}, (\pi, \sigma, i, j) \models \varphi$").

[4] Note that this allows for flexible functions. Therefore, constants correspond to flexible state variables. However, we will not exploit the generality of this concept and assume that the interpretation of n-ary function symbols with $n > 0$ is the same across all states.

i) $\mathcal{M},(\pi,\sigma,i,j) \models AG\ \varphi$ iff $\forall\sigma' \in \pi.$
$(\sigma^{0,j} < \sigma'$ implies $\forall k \geq j.\mathcal{M},(\pi,\sigma',i,k) \models \varphi)$

ii) $\mathcal{M},(\pi,\sigma,i,j) \models AF\ \varphi$ iff $\forall\sigma' \in \pi.$
$(\sigma^{0,j} < \sigma'$ implies $\exists k \geq j.\mathcal{M},(\pi,\sigma',i,k) \models \varphi)$

iii) $EF\ \varphi =_{Df} \neg AG\ \neg\varphi$

iv) $\mathcal{M},(\pi,\sigma,i,j) \models\ \blacksquare\ \varphi$ iff $\forall k.(i \leq k \leq j$ implies $\mathcal{M},(\pi,\sigma,i,k) \models \varphi)$

v) $\blacklozenge\ \varphi =_{Df} \neg\ \blacksquare\ \neg\varphi$

vi) $\mathcal{M},(\pi,\sigma,i,j) \models [\varphi]$ iff $\mathcal{M},(\pi,\sigma,i,j) \models \varphi$
and $(j > 0$ implies $M,(\pi,\sigma,i,j-1) \not\models \varphi)$

vii) $\mathcal{M},(\pi,\sigma,i,j) \models \#[\varphi] =_{Df}$

$$\begin{cases} 0 & \text{if} \quad j = i \text{ and } \mathcal{M},(\pi,\sigma,i,j) \models \neg[\varphi] \\ 1 & \text{if} \quad j = i \text{ and } \mathcal{M},(\pi,\sigma,i,j) \models [\varphi] \\ \mathcal{M},(\pi,\sigma,i,j-1) \models \#[\varphi] & \text{if} \quad j > i \text{ and } \mathcal{M},(\pi,\sigma,i,j) \models \neg[\varphi] \\ \mathcal{M},(\pi,\sigma,i,j-1) \models \#[\varphi]+1 & \text{if} \quad j > i \text{ and } \mathcal{M},(\pi,\sigma,i,j) \models [\varphi] \end{cases}$$

The future operators AG, AF, and EF are interpreted as usual:

- $AG\ \varphi$ (read "φ is global") means that φ is true now and will hold in all possible future states.
- $AF\ \varphi$ (read "φ is inevitable") means that φ holds now or will hold for every possible future at some moment in time.
- $EF\ \varphi$ (read "φ is potential") that φ holds now or will hold for some possible future at some moment in time.

The meaning of the past operators $\blacksquare$ and $\blacklozenge$ covers the general case where a context in the past has been established using interval operators:

- $\blacksquare\ \varphi$ (read "always in the past φ") means that φ is true now and has always been true in the current context in the past.
- $\blacklozenge\ \varphi$ (read "sometimes in the past φ") means that φ is true now or has been true sometime in the current context in the past.

Furthermore, there are past operators referring to actions, which are interpreted as follows:

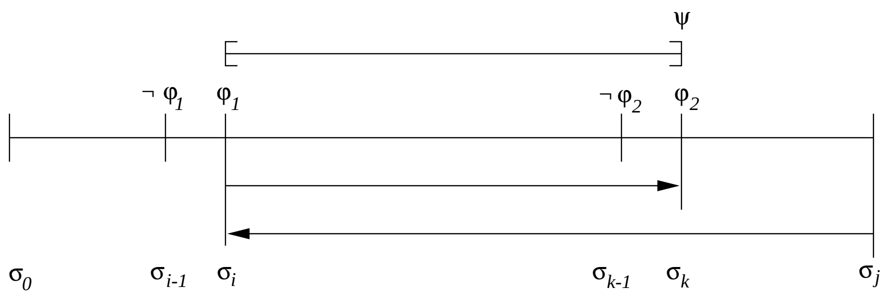
- $[\varphi]$ (read "action of type φ") means that φ has just become true.
- $\#[\varphi]$ (read "number of actions of type φ") denotes the number of actions of type φ in the current context in the past.

The semantics of interval operators is a little involved, therefore, we give an informal idea about their interpretation. For a complete formal semantics, the reader is referred to [7].

To construct intervals, the notion of action is used. With an action, an atomic interval, i.e., a single point in time, is associated. Complex intervals can be constructed with the help of two interval operators $\Rightarrow$ and $\Leftarrow$. Intuitively, the directions of the arrows indicate in which direction and in which order the interval end points are located. The end point of the interval expressed by the interval term at the end of the arrow is located first, followed by a search, in the direction of the arrow, for the end point of the interval denoted by the interval term at the head of the arrow. An explicit operator $\bullet$ to refer to the next/last but one action is introduced.

- $\langle\beta\rangle\varphi$ (read "φ restricted on the context β") means that φ holds in the context established by the interval β. It is vacuously true if the context does not exist.
- $\bullet\alpha$ (read "the last but one action α" and "the next but one action α", respectively) means that if the search direction is backward, $\bullet\alpha$ refers to the last but one action ; if it is forward, $\bullet\alpha$ denotes the next but one action.
- *begin* δ and *end* δ denote atomic intervals consisting of the first and the last state of the interval δ, respectively.
- $\delta_1 \Rightarrow \delta_2$ denotes the interval commencing at the end of δ_1 and ending at the end of δ_2, where δ_1 is located first, and δ_2 is searched within the current context in the forward direction, starting with the end of δ_1.
- $\delta_1 \Leftarrow \delta_2$ denotes the interval commencing at the end of δ_1 and ending at the end of δ_2, where δ_2 is located first, and δ_1 is searched within the current context in the backward direction, starting with the end of δ_2.

Example: Let φ_1, φ_2, ψ be formulas, $\sigma^{0,j}$ be a state sequence representing the current context in the past. The interpretation of $\langle[\varphi_1] \Rightarrow [\varphi_2]\rangle\psi$ proceeds as illustrated below: starting from the current state σ_j, the end point of the interval denoted by $[\varphi_1]$ is located first, searching backward. In the diagram, this end point is state σ_i. Next, the end point denoted by $[\varphi_2]$ is located, searching in the direction of the arrow, i.e. forward. In the diagram, this leads to σ_k. Thus, the new current context is evaluated as $\sigma^{i,k}$. Formula ψ is now evaluated in the last state of this context. In case of an open ended arrow, the corresponding end point coincides with the boundary of the search context. For instance, the interpretation of $\langle[\varphi_1] \Rightarrow\rangle\psi$ would yield the new current context $\sigma^{i,j}$.



When we use temporal logic to characterize a system, we require that a specification must hold in the *initial* state of execution (properties holding throughout the execution can be defined using the operator AG). To express this formally, we use the notion of initial validity. A formula φ is *initially-satisfied in a model*

$\mathcal{M}$ *for a tree* $\pi \in \Pi$, written $\mathcal{M}, \pi \models_i \varphi$, iff $\mathcal{M}, (\pi, \sigma, 0, 0) \models \varphi$ is true, where $\sigma \in \pi$. φ is *initially-valid in* $\mathcal{M}$ iff φ is initially-satisfied for all $\pi \in \Pi$. Finally, φ is *initially-valid*, written $\models_i \varphi$, iff φ is initially-valid in all models $\mathcal{M}$.

2.4 System

Having introduced system architectures and system behaviours, we can combine them into the derived concept of *system*.

Definition 4 (system). A system $\mathcal{S}$ is a structure $\langle Arch, Behav \rangle$, where $Arch$ is a system architecture and $Behav$ is a system behaviour.

2.5 System refinement

A notion of system refinement should take both system architecture and system behaviour into account. In general, it is desirable that the refinement of a single component (architecture and/or behaviour) has no influence on the other components. Only then will it be possible to perform incremental system design and modular verification, which is a prerequisite for the development of large systems. By incremental system design, we mean that we can modify or replace a part of the system without affecting the other parts. Modular verification means that only the modified or replaced parts have to be verified, not the entire system. To allow for incremental system design and modular verification, we have to make suitable restrictions with respect to architecture and behaviour.

A system $\mathcal{S}'$ *refines* a system $\mathcal{S}$, if $\mathcal{S}'$ is equivalent to or more specific than $\mathcal{S}$. With respect to the system architecture, this is the case if all agents and interaction points of $\mathcal{S}$ are represented in $\mathcal{S}'$, and if their composition is maintained in $\mathcal{S}'$. This is captured by the notion of architectural refinement below. Concerning the behaviour, $\mathcal{S}'$ is equivalent to or more specific than $\mathcal{S}$ if the behaviour of $\mathcal{S}'$ logically implies the behaviour of $\mathcal{S}$ mapped to the abstraction level of $\mathcal{S}'$. This mapping is expressed by a *representation function rep*.

Definition 5 (system refinement). Let $\mathcal{S}= \langle Arch, Behav \rangle$ and $\mathcal{S}' = \langle Arch',$ $Behav' \rangle$ be systems. $\mathcal{S}'$ is a refinement of $\mathcal{S}$ under the representation function rep (written "$\mathcal{S}'\ refines_{rep}\ \mathcal{S}$") if and only if the following is satisfied[5]:

- $Arch'\ refines\ Arch$
- $\models_i Behav' \supset rep(Behav)$

When we use temporal logic to characterize system behaviour, we require that a property must hold in the *initial* state of execution (properties holding throughout the execution can be defined using appropriate temporal operators). To express this formally, we use the notion of initial validity. Therefore, it is sufficient to require $Behav' \supset rep(Behav)$ to be initially-valid.

With respect to architectures, we require that agents and interaction points be refined separately. In other words, a single component of the refinement (an

[5] *Refinement* as defined here has been termed *conformance* in [7].

agent or interaction point) is uniquely related to a single component of the refined architecture. Also, we require that the number of interaction points an agent is associated with remains the same. These and further architectural constraints can be formalized as follows:

Definition 6 (architectural refinement). Let $Arch = \langle Ag, Ip, ArchF \rangle$ and $Arch' = \langle Ag', Ip', ArchF' \rangle$ be architectures. $Arch'$ is an *architectural refinement* of $Arch$ (written "$Arch'$ $refines$ $Arch$") if and only if there is a refinement function $ref : Ag \cup Ip \rightarrow 2^{Ag' \cup Ip'}$ such that the following restrictions hold:

- Each component of $Arch$ is refined, i.e., ref is a total function.
- The refinement of an agent must include at least one agent. Formally: $\forall ag \in Ag.\ ref(ag) \cap Ag' \neq \{\}$.
- The refinement of an interaction point must include at least one interaction point: $\forall ip \in Ip.\ ref(ip) \cap Ip' \neq \{\}$.
- Each agent and each interaction point is refined separately, i.e., the refinement is disjoint: $\forall x, y \in Ag \cup Ip.\ (x \neq y \ implies \ ref(x) \cap ref(y) = \{\})$.
- Ag' is the set of exactly those agents resulting from the refinement, i.e., $Ag' = (\bigcup_{ag \in Ag} ref(ag) \cup \bigcup_{ip \in Ip} ref(ip)) \setminus Ip'$.
- Ip' is the set of exactly those interaction points resulting from the refinement, i.e., $Ip' = (\bigcup_{ip \in Ip} ref(ip) \cup \bigcup_{ag \in Ag} ref(ag)) \setminus Ag'$.
- If an agent $ag \in Ag$ is associated with an interaction point $ip \in Ip$, then exactly one agent of the refinement of ag must be associated with exactly one interaction point of the refinement of ip. Formally:

  $\forall ip \in Ip.\ \forall ag \in Ag.\ ip \in ArchF(ag)\ implies$

  $$(\mid ref(ip) \setminus Ag' \cap \bigcup_{ag' \in ref(ag) \setminus Ip'} ArchF'(ag') \mid = 1 \ and$$
  $$\mid \{ag' \in ref(ag) \setminus Ip' \mid ArchF'(ag') \cap ref(ip) \setminus Ag' \neq \{\} \} \mid \ = 1)$$

Figure 2 shows the graphical representation of a possible refinement of the interaction point ip (compare Figure 1), which on a lower level of abstraction comprises an agent ag that can interact with ag_1, ag_2, and ag_3 through ip_1, ip_2, and ip_3, respectively. It is necessary to introduce interaction points in the refinement, because otherwise the rule of composition of architectures about their explicit introduction would be violated. This rule supports the composition of system behaviour from component behaviour. Also, we notice that the duality between agents and interaction points is nicely carried into the refinement.

3 The memory component

In this section, we will apply the formalism outlined in Section 2 to the unreliable and the reliable memory component of the RPC-Memory Specification Problem. We will explain why the reliable memory component implements the unreliable memory, and how the memory can be architecturally refined.

The general concepts introduced in Section 2 can be related to the domain-specific concepts of the RPC-Memory Specification Problem as shown in Table 1.

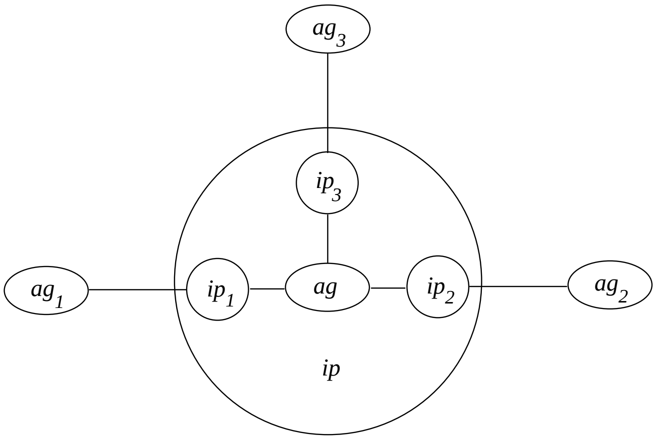


Fig. 2. Graphical representation of architectural refinement

3.1 A temporal logic

To specify the memory component, we use the logic defined in Section 2.3. We introduce sort symbols *Ag*, *Ip*, *Ao*, *Pn*, *Proc*, *Arg*, $RetVals$, $MemLocs$, and $MemVals$ interpreted as sets of agents (here: components), interaction points, abstract operations, interaction types (here: procedure names), processes, argument lists, return values, memory locations, and memory values, respectively. Furthermore, we define constants M, *Env* (of sort *Ag*), *ip* (of sort *Ip*), !, ? (of sort *Ao*, denoted *offer* and *accept*), R, W (of sort *Pn*), $InitVal$ (of sort $MemVals$) and *BadArg*, $Fail$, OK (of sort $RetVals$). Individual variables are *pn*, pn_1, pn_2 (of sort *Pn*), *p*, p_1, p_2, p', p'' (of sort *Proc*), *a*, a_1, a_2, *l*, *v* (of sort *Arg*), *r*, r', r'', r_1, r_2 (of sort $RetVals$), and n_1, n_2 (of sort $\mathbb{N}_0$). There is one function symbol *ArchF*, interpreted as the architecture function and associated with sorts 2^{Ip} and *Ag*. at_i and $after_i$, for $i \in \{3, 4, 5, 6, 7\}$, are relation symbols of arity i, associated with i sorts *Ag*, *Ip*, *Ao*, *Pn*, *Proc*, *Arg*, $RetVals$ (in that order).

general concept	**domain-specific concept**
agent	component, process
interaction point	(procedure-calling) interface
interaction	call, return

Table 1. Relationship between general and domain-specific concepts

Some restrictions as explained below apply with respect to their interpretation. In the specification of the memory component, we will omit the index i.

An interaction may occur between two agents at a common interaction point and consists of offer and acceptance, denoted by the abstract operations ! and ?, respectively. It has an interaction type pn (e.g., R for *Read* or W for $Write$) and interaction parameters (e.g., arguments and return values). We write $ag.ip.!\, pn(p,a,r)$ to denote that the agent ag offers an interaction of type pn with parameter values p, a, r at interaction point ip. Similarly, $ag.ip.?\, pn(p,a,r)$ denotes that agent ag accepts an interaction, where pn, p, a, and r will have values of a previous interaction offer. A procedure call can be considered as one particular interaction scheme: call and return are modeled as different interactions, each consisting of offer and acceptance. Note that the concepts of call and return are different from the concepts of interaction offer and acceptance, therefore, they do not correspond to the abstract operations ! and ?.

The formula $at_7\ ag.ip.!\, pn(p,a,r)$ (i.e. $(ag, ip, !, pn, p, a, r) \in at_7$) holds when ag is prepared to offer an interaction of type pn with parameter values p, a, r at interaction point ip; $after_7\ ag.ip.!\, pn(p,a,r)$ holds immediately after completion of the offer. We require that any interpretation of at_7 and at_6 satisfies $AG\ \forall pn, p, a, r.\ (at_7\ ag.ip.!\, pn(p,a,r) \supset at_6\ ag.ip.!\, pn(p,a))$, so the formula $at\ ag, ip.!\, pn(p,a)$ holds when the agent ag is prepared to offer an interaction of type pn with parameter values p, a and some additional parameter value r at ip. Similar constraints apply to at_i and $after_i$, for $i \in \{3,4,5,6,7\}$, and can be stated formally. Since the index i is clear from the context, it will be omitted in the following specification. Also, we omit $after$ and ip.

With these preparations, we can now give a requirement specification of the memory component.

3.2 Specification of the unreliable memory component

The specification of the unreliable memory component is listed in Table 2. As compared to the version presented in Dagstuhl, several improvements have been made to solve problems pointed out by Reino Kurki-Suonio, Leslie Lamport and Stephan Merz. All these versions have avoided the use of process identities for procedure calls. Finally, further problems discovered by Reino Kurki-Suonio have led to the present version which makes use of process identities (which is in line with the problem statement) in order to establish a relationship between calls and returns.

The architecture consists of the memory component M, the environment *Env* and an interaction point ip. The behaviour of M is given as the conjunction of properties P_1 through P_6, the interaction point behaviour is captured by properties P_7 through P_9. No properties are specified for the environment, which means that its behaviour is not restricted. All specified properties will be explained informally below. Additionally, some restrictions are placed on the sets of memory locations, memory values, arguments, and return values.

The first part of property P_1 specifies that for each process p and argument a, the number of $Read/Write$ returns must not exceed the number of $Read/Write$

calls. More technically, the number of actions where M has offered an interaction of type $Read/Write$ with parameter values p and a at interaction point ip (i.e. returns) is less or equal to the number of actions where M has accepted a corresponding interaction at ip (i.e. calls). This captures part of the semantics of the procedure calling mechanism from M's point of view. The second part of property P_1 states that for each process p, only one procedure call can be pending at any time.

Next, P_2 states that if $BadArg$ is returned as the result of a $Read/Write$ call, then the argument is not in the proper domain, therefore an exception is raised.

P_3 reads as follows: if value r is returned as the result of a $Read$ call of process p on location l, one of the following must hold:

- $r \in \{BadArg, Fail\}$, or
- $r = InitVal$, and no $Write$ into location l has ever ended successfully, or
- a successful $Write$ into location l has occurred in the past before the $Read$ call corresponding to the $Read$ return in the antecedents, and some $Write$ of value r into location l has been pending at some point in time since the call of the successful $Write$, or
- no successful $Write$ into location l has occurred in the past before the $Read$ call corresponding to the $Read$ return in the antecedents, and some $Write$ of value r into location l has been pending at some point in time.

The interval constructed in P_3 needs some explanation: starting from the current state, the starting point denoted by the expression $(([\varphi_1] \Leftarrow [\varphi_2]) \Leftarrow [\varphi_3])$ is located by searching backward first for $[\varphi_3]$, then for $[\varphi_2]$, and finally for $[\varphi_1]$.

If some $Write$ of value r into location l is pending, an atomic write into that location may occur. Whether it actually *does* occur may only become externally visible through the completion of a pending or subsequent $Read$. It is, however, possible that atomic writes may no longer become externally visible, namely if there is a successful $Write$ afterwards. The precise characterization of this scenario is given in the third alternative of P_3.

Property P_4 specifies an additional constraint to exclude illegal behaviour not ruled out by P_3. Consider the following scenario: two subsequent successful $Read$ operations on location l are performed such that no atomic writes may have changed the value of the memory cell between the corresponding atomic reads. Then both $Read$ operations must return the same value, unless there is some memory failure which may result in the return value $Fail$. This condition is stated in two parts, distinguishing between possible orderings of the $Read$ calls (for all processes p' and return values r'):

- If a $Read$ call and a return of process $p' \neq p$ have occurred since the last $Read$ call of process p, then the returned values are the same, $Fail$ is returned, or some atomic write may have happened in that interval.
- If no $Read$ call and return of process p' have occurred since the last $Read$ call of process p, but before the current $Read$ return of process p, then again

$\mathcal{S} = \langle Arch, Behav \rangle$

$Arch = \langle \{M, Env\}, \{ip\}, ArchF \rangle$ with $ArchF(M) = ArchF(Env) = \{ip\}$

$Behav = \bigwedge_{1 \leq i \leq 10} P_i$

$MemLocs \cup (MemLocs \times MemVals) \subseteq Arg$
$BadArg \notin MemVals \wedge Fail \notin MemVals$
$MemVals \subseteq RetVals$

P_1. $AG\ \forall p, a.\ (\#[M.?\ R(p,a)] \geq \#[M.!\ R(p,a)]\ \wedge\ \#[M.?\ W(p,a)] \geq \#[M.!\ W(p,a)]) \wedge$
$AG\ \forall p.\ (0 \leq \#[M.?\ W(p)] + \#[M.?\ R(p)] - \#[M.!\ W(p)] - \#[M.!\ R(p)] \leq 1)$

P_2. $AG\ \forall p, a.\ ([M.!\ R(p, a, BadArg)] \supset a \notin MemLocs) \wedge$
$AG\ \forall p, a.\ ([M.!\ W(p, a, BadArg)] \supset a \notin MemLocs \times MemVals)$

P_3. $AG\ \forall p, l, r.\ ([M.!\ R(p, l, r)]$
$\supset (r \in \{BadArg, Fail\} \vee$
$r = InitVal\ \wedge\ \neg\ \exists p', r'.\ \blacklozenge\ [M.!\ W(p, (l, r'), OK)] \vee$
$\blacklozenge\ ([M.?\ R(p,l)]\ \wedge\ \exists p', r'.\ \blacklozenge\ [M.!\ W(p', (l, r'), OK)]) \wedge$
$\exists p', r'.\ \langle begin\ (([M.?\ W(p', (l, r'))] \Leftarrow [M.!\ W(p', (l, r'), OK)]) \Leftarrow [M.?\ R(p,l)]) \Rightarrow \rangle$
$\exists p'.\ \blacklozenge\ (\#[M.?\ W(p', (l, r))] > \#[M.!\ W(p', (l, r))])) \vee$
$\neg\ \blacklozenge\ ([M.?\ R(p,l)]\ \wedge\ \exists p', r'.\ \blacklozenge\ [M.!\ W(p', (l, r'), OK)]) \wedge$
$\exists p'.\ \blacklozenge\ (\#[M.?\ W(p', (l, r))] > \#[M.!\ W(p', (l, r))])))$

P_4. $AG\ \forall p, p', l, r, r'.\ ([M.!\ R(p, l, r)] \wedge r \notin \{BadArg, Fail\}$
$\supset \langle [M.?\ R(p,l)] \Rightarrow \rangle\ (p' \neq p\ \wedge\ \blacklozenge\ ([M.!\ R(p', l, r')]\ \wedge\ \blacklozenge\ [M.?\ R(p', l)])$
$\supset (r' \in \{r, Fail\}\ \vee\ \blacklozenge\ \exists p''.\ (\#[M.?\ W(p'', (l, r))] > \#[M.!\ W(p'', (l, r))]))) \wedge$
$\langle [M.?\ R(p,l)] \Rightarrow \rangle\ \neg\ (p' \neq p\ \wedge\ \blacklozenge\ ([M.!\ R(p', l, r')]\ \wedge\ \blacklozenge\ [M.?\ R(p', l)]))$
$\supset \langle begin([M.?\ R(p', l)] \Leftarrow [M.!\ R(p', l, r')]) \Rightarrow \rangle$
$(r' \in \{r, Fail\}\ \vee\ \blacklozenge\ \exists p''.\ (\#[M.?\ W(p'', (l, r))] > \#[M.!\ W(p'', (l, r))])))$

P_5. $AG\ \forall pn, p.\ (\#[M.?\ pn(p)] > \#[M.!\ pn(p)] \supset AF\ at\ M.!\ pn(p))$

P_6. $AG\ \forall pn, p, a.\ \exists r.\ ([M.?\ pn(p, a)] \supset EF\ (at\ M.!\ pn(p, a, r) \wedge r \neq Fail))$

P_7. $AG\ \forall pn_1, pn_2, p_1, p_2, a_1, a_2, r_1, r_2, n_1, n_2.$
$((\#[M.?\ pn_1(p_1, a_1)] = n_1 \wedge \#[M.?\ pn_2(p_2, a_2)] = n_2$
$\supset \blacklozenge\ (\#[Env.!\ pn_1(p_1, a_1)] = n_1 \wedge \#[Env.!\ pn_2(p_2, a_2)] = n_2)) \wedge$
$((\#[Env.?\ pn_1(p_1, a_1, r_1)] = n_1 \wedge \#[Env.?\ pn_2(p_2, a_2, r_2)] = n_2$
$\supset \blacklozenge\ (\#[M.!\ pn_1(p_1, a_1, r_1)] = n_1 \wedge \#[M.!\ pn_2(p_2, a_2, r_2)] = n_2)))$

P_8. $AG\ (at\ M.! \wedge at\ Env.? \supset AF\ [M.!] \wedge AF\ [Env.?]) \wedge$
$AG\ (at\ Env.! \wedge at\ M.? \supset AF\ [Env.!] \wedge AF\ [M.?])$

P_9. $AG\ (at\ M.? \wedge \#[Env.!] > \#[M.?] \supset AF\ [M.?]) \wedge$
$AG\ (at\ Env.? \wedge \#[M.!] > \#[Env.?] \supset AF\ [Env.?])$

Table 2. Specification of the unreliable memory component

the returned values are the same, $Fail$ is returned, or some atomic write may have happened in the interval starting with the earlier *Read* call.

As P_1 through P_3, P_4 is a safety property. Properties P_5 and P_6 are liveness properties stating that

- for each $Read/Write$ call of process p, a corresponding $Read/Write$ return is eventually offered (P_5), and
- a $Read/Write$ call potentially leads to a return with a return value other than $Fail$ (P_6).

P_6 is the reason why we have used a branching time temporal logic. With linear time, we could, for instance, specify strong fairness with respect to a return value other than $Fail$, which in our opinion would solve the issue raised in Problem 1c. P_6 is a different solution to this issue, since it is orthogonal to strong fairness: systems satisfying P_6 may not implement strong fairness, and vice versa.

The specification of the system behaviour is not complete without the behaviour of the interaction point *ip* [5]. So far, we have addressed the ability and readiness of the memory component M to interact with its environment *Env* through *ip*. It is still left open under what circumstances interactions can occur. E.g., interactions can be synchronous, in which case offer and acceptance must occur simultaneously, or asynchronous, meaning that the offer occurs before the acceptance. Synchronous interaction is a frequent assumption for process algebras, while asynchronous interaction is typical for communicating state machines. In an early development stage, it may also be left open whether interactions occur synchronously or asynchronously. Another difference is whether an interaction offered at some point in time may "cancel" an interaction offered earlier, but not yet accepted, or whether interactions offered are always kept until they are accepted. The first case can be related to the concept of a shared variable with a single value, while in the second case, we may think of a set or - more specifically - a queue.

We require that only interactions that have been offered by *Env* (M) can be accepted by M (*Env*). In addition, we require that the order of interactions is maintained (FIFO-property). However, no restriction is made whether interactions between M and *Env* have to occur synchronously or asynchronously. Thus, a decision for one particular form of interaction can be made in a later design stage. The above requirements are captured by property P_7. Note that P_7 is a safety property, expressing that for all combinations of interaction types pn_1, pn_2, and parameter vectors p_1, a_1, p_2, a_2, if at some point in time the interactions $pn_1(p_1, a_1)$ and $pn_2(p_2, a_2)$ have been accepted n_1 times and n_2 times at *ip*, respectively, then there must be some previous point in time where $pn_1(p_1, a_1)$ and $pn_2(p_2, a_2)$ have been offered n_1 times and n_2 times, respectively (similar for $pn_1(p_1, a_1, r_1)$ and $pn_2(p_2, a_2, r_2)$) This is provably equivalent to the FIFO-property (for a proof, see [5]).

Since P_7 is already satisfied in case of inaction, it should be complemented by liveness requirements. Property P_8 expresses that if M is prepared to offer and

Env is prepared to accept an interaction, then both agents will eventually proceed. Progress does not have to occur simultaneously, and may concern different interactions. Again, it is left open whether interactions occur synchronously or asynchronously. Therefore, P_8 should be augmented by P_9, which expresses that if more interactions have been offered than accepted, and M (*Env*) is prepared to accept, then it will eventually proceed. Interestingly, P_7, P_8, and P_9 together prevent loss of interactions: if an interaction would be lost, this would violate P_7, because to satisfy P_7 and P_8, an interaction would have to be created.

3.3 Refinements of the unreliable memory component

In this section, the unreliable memory component will be refined in two respects. Firstly, we will give a specification of a reliable memory component and argue that this is a system refinement (see Section 2.5). Secondly, we will make one step in the direction of a distributed implemention of the unreliable memory component by refining the architecture.

The specification of the reliable memory $\mathcal{S}'$ is identical with the specification of the unreliable memory component $\mathcal{S}$ (see Table 2) with the exception of properties P_3 and P_6. In P_3, $Fail$ has to be removed, since in the reliable memory, this is no longer a possible return value. In P_6, the temporal operator EF has to be replaced by AF, because the reliable memory inevitably has to return a value other than $Fail$. For better readability, the second part of the conjunction, i.e. $r \neq Fail$, could be removed, since that is already covered by $P3$.

Proposition 7. *Let rep be the identical mapping between formulas, i.e.* $\forall\, \varphi.\ rep(\varphi) = \varphi$. *Then the reliable memory* $\mathcal{S}' = \langle Arch', Behav' \rangle$ *is a refinement of the unreliable memory* $\mathcal{S} = \langle Arch, Behav \rangle$ *under rep, i.e.* $\mathcal{S}'\ refines_{rep}\ \mathcal{S}$.

Proof. With the definition of system refinement in Section 2.5, there are two proof obligations:

a) to show: $Arch'\ refines\ Arch$

   Define ref to be the identical mapping between architectures, i.e. $\forall\, x \in Ag \cup Ip.\ ref(x) = \{x\}$.

b) to show: $\models_i Behav' \supset rep(Behav)$

   With rep as defined, and $\models \varphi$ implies $\models_i \varphi$, it suffices to show: $\models Behav' \supset Behav$. With $Behav$ and $Behav'$ as given, this follows from $\models P_3' \supset P_3$ and $\models P_6' \supset P_6$:

   i) $\models P_3' \supset P_3$

      $AG\ \forall p, l, r.\ ([M.!\ R(p,l,r)]\ \supset (r \in \{BadArg\}\ \vee\ ...))$

      implies[6]

      $AG\ \forall p, l, r.\ ([M.!\ R(p,l,r)] \supset (r \in \{BadArg\}\ \vee\ r \in \{Fail\}\ \vee\ ...))$

[6] due to $\models (\varphi \supset \varphi') \supset (\varphi \supset (\varphi' \vee \varphi''))$

iff[7]
$AG\ \forall p, l, r.\ ([M.!\ R(p, l, r)] \supset (r \in \{BadArg, Fail\}\ \vee\ ...))$

ii) $\models P_6' \supset P_6$
$AG\ \forall pn, p, a.\ \exists r.\ ([M.?\ pn(p, a)] \supset AF\ (at\ M.!\ pn(p, a, r)\ \wedge\ r \neq Fail))$
implies[8]
$AG\ \forall pn, p, a.\ \exists r.\ ([M.?\ pn(p, a)] \supset EF\ (at\ M.!\ pn(p, a, r)\ \wedge\ r \neq Fail))$

q.e.d.

Next, we refine the memory component M into a reliable memory component RM, an RPC component, a clerk C and two interaction points ip_1 and ip_2. Formally, we get the architecture $Arch'' = \langle \{RM, RPC, C, Env\}, \{ip, ip_1, ip_2\}, ArchF'' \rangle$ with $ArchF''(RM) = \{ip_2\}$, $ArchF''(RPC) = \{ip_1, ip_2\}$, $ArchF''(C) = \{ip_1, ip\}$, and $ArchF''(Env) = \{ip\}$ (see Figure 3). With the refinement function $ref(M) = \{RM, RPC, C, ip_1, ip_2\}$, $ref(Env) = \{Env\}$, $ref(ip) = \{ip\}$, this qualifies as a legal architectural refinement, i.e. $Arch''\ refines\ Arch$.

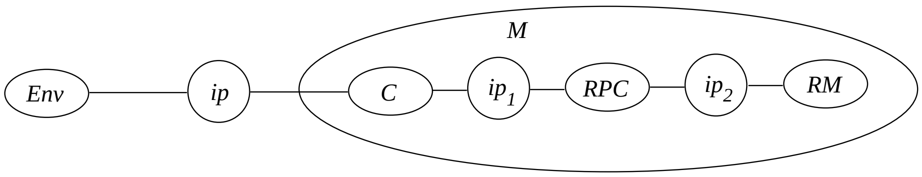


Fig. 3. Graphical representation of the refined memory architecture $Arch''$

Some comments are in place here. Although it is technically feasible to refine M as described, and to specify the behaviour of the components in a property-oriented way, we would prefer a different approach in this particular case. The reason for this is that the refinement as suggested in the problem statement introduces a system structure which in our opinion is not a requirement, but one of several possible design decisions, and which therefore should not be prescribed. It would be sufficient here to respect the original architecture consisting of M, Env and ip, and to introduce the refined structure in a more informal way, e.g., in order to improve the readability of the specification. This, however, is not supported by our formalism. Also, the refinement of M is given in operational terms in the problem statement. Thus, it would be more appropriate to use an

[7] due to $\models \forall x.\ ((x \in A \vee x \in B) \equiv (x \in A \cup B))$
[8] due to $\models AF\ \varphi \supset EF\ \varphi$

operational specification style together with a specification structure that does not express an architectural requirement in this particular case.

4 Are architectural requirements important?

It has been argued that architectural requirements are not essential and should therefore be omitted. This argument manifests itself in equivalence or correctness notions abstracting away from specification structure. A typical example here are LOTOS specifications which may be composed from a number of LOTOS processes [3]. Transformations may be such that the specification structure is changed, while the visible behaviour remains the same, thus resulting in equivalent specifications. E.g., a monolithic specification consisting of a single LOTOS process may be equivalent to a resource oriented specification composed of several processes [9]. This way, structure is exploited to improve readability and modularity of a specification. However, if specifications with different structure can be equivalent, the structure does not have a formal semantics and therefore does not form an architectural requirement. Consequently, architectural requirements cannot be expressed in languages like LOTOS. If they are considered essential, they can only be dealt with informally.

Other examples where the structure of a specification has no formal semantics can be found in this volume (e.g., see [1], [2], [4], [8]). In these cases, structure is exploited to achieve better readability, and to support the verification process. Also, it can be observed that the specification structure coincides with the structure suggested in the informal problem statement. However, it does not form a requirement and may be modified by the system designer in a later refinement step.

In Section 2.5, we have introduced a correctness notion respecting architectural requirements. Recall that one condition for a system $\mathcal{S}'$ refining a system $\mathcal{S}$ is that all agents and interaction points of $\mathcal{S}$ are represented in $\mathcal{S}'$, and that their composition is maintained in $\mathcal{S}'$. Consequently, systems with different architectures cannot be equivalent, or, stated otherwise, system architecture forms a requirement.

While we agree with the view that the structure of a specification may serve better readability and modularity, we argue that in certain cases, structure carries essential information about the specified system and therefore should have a formal semantics. E.g., consider a communication service with two service users ag_1 and ag_2 exchanging messages via interaction points ip_1 and ip_2. When ag_1 sends a message at ip_1, this message should be delivered to ag_2 through ip_2. However, any system delivering this message back to ag_1 through ip_1 would obviously not be correct. Abstracting away from the system architecture would prevent us from distinguishing the two cases, therefore, architectural reference points are necessary if such a distinction is considered important. Distinctions of this kind are typically drawn in the area of distributed systems when distribution is a requirement as in the case of a communication system.

In the example of the communication service, it may be argued that one can make a distinction by introducing additional message parameters, e.g. a destination address. However, this can be seen as just another way of introducing architectural aspects, with the clear disadvantage that no formal architectural meaning is associated with these parameters. Instead, this approach heavily relies on background knowledge not expressed in the specification and therefore is an informal way of treating architectural aspects.

In summary, architectural requirements are important in some cases and therefore should be expressable. This creates reference points that can be used when specifications on different abstraction levels have to be related, e.g., for verification purposes. On the other hand, a specification structure may just serve better readability and modularity, and does not form a requirement for the implementor. A specification technique for distributed systems should be able to distinguish between structure forming a requirement and structure that does not. If structure is important, it should be possible to express this explicitly and formally. If it is not important, it should not be a requirement.

5 Conclusion

The RPC-Memory Specification Problem partially treated in this report can be classified as an informal design specification. It is largely operational and relies on an intuitive understanding of well-known mechanisms, such as procedure calls and the reading and writing of memory cells. This makes it well suited for techniques directed towards design specifications, i.e. constructive FDTs that are able to model these mechanisms by closely following structure and style of the informal specification and then putting the parts together. However, the formalism applied above is more suited for requirement specifications which usually are more abstract. The goal here has been to capture the externally visible behaviour of each component by a conjunction of logical formulas. Therefore, our first step has been to fathom the external behaviour implicitly stated by the informal specification, and, as the second step, to capture it by logical properties. We have limited confidence that the resulting specification does not exclude legal memory behaviour. Also, we are not completely convinced that all illegal behaviour according to the problem statement is ruled out. The reason is that we had to insert a reengineering step in order to get from the operational style to the property style, a step which certainly must be considered as extremely errorprone. However, if we understand our result as a requirement specification, i.e. a specification that does not rule out legal behaviour – in contrast to a design specification capturing legal behaviour only –, then that result appears in a different light.

Acknowledgements. Special thanks go to Reino Kurki-Suonio and Stephan Merz, who have provided very detailed, constructive comments and extremely valuable feedback.

References

1. M. Abadi, L. Lamport and S. Merz. *The Dagstuhl Problem – TLA+ Solution*, this volume
2. J. Blom and B. Jonsson. *Architecture Oriented Temporal Logic Specification*, this volume
3. T. Bolognesi and E. Brinksma. *Introduction to the ISO Specification Language LOTOS*, Computer Networks, 14(1), Jan. 1989
4. M. Broy. *A Functional Solution to the Memory and RPC Specification Problem*, this volume
5. R. Gotzhein. *Formal Definition and Representation of Interaction Points*, Computer Networks and ISDN Systems 25 (1992) 3-22
6. R. Gotzhein. *Temporal Logic and Applications – A Tutorial*, Computer Networks and ISDN Systems 24 (1992) 203-218
7. R. Gotzhein. *Open Distributed Systems - On Concepts, Methods and Design from a Logical Point of View*, Vieweg Wiesbaden, 1993
8. K. G. Larsen, B. Steffen and C. Weise. *A Constraint Oriented Proof Methodology*, this volume
9. C. A. Vissers, G. Scollo and M. van Sinderen. *Architecture and Specification Style in Formal Descriptions of Distributed Systems*, in: S. Aggarwal, K. Sabnani (eds.), Protocol Specification, Verification, and Testing VIII, Proceedings, Atlantic City, USA, June 7-10, 1988, 16p.