The RPC-Memory Case Study: A Synopsis *

Manfred Broy, Stephan Merz, and Katharina Spies

Institut für Informatik, Technische Universität München
Arcisstr. 21, 80290 München, Germany

1 About this book

The RPC-Memory specification problem was proposed by Broy and Lamport as a case study in the formal design of distributed and concurrent systems. The idea was to use it as a basis for comparing various approaches to formal specification, refinement, and verification. Various preliminary solutions were presented and discussed during a workshop at Schloß Dagstuhl, Germany, in September 1994. Authors were then given the opportunity to revise their specifications to reflect the discussions at the seminar. An extensive refereeing process ensured and authors were encouraged to discuss their solutions with the referees.

This volume contains fifteen solutions to the RPC-Memory problem that resulted from this process. The formalisms that underly the specifications reflect different schools of system specification, including Petri nets, temporal and higher-order logics, various formats of transition systems or automata, and stream-based approaches, supporting various degrees of formalized or computer-assisted verification. The contributors were free to solve only those aspects of the problem that they considered particularly important or omit aspects that could not be adequately represented in the chosen formalism.

Section 2 of this introductory overview reviews the specification problem, discussing its structure and the problems posed to the participants. In section 3 we attempt to classify the solutions contained in this volume. We indicate which parts of the problem have been addressed and what we believe to be the key points of each solution. This would have been impossible without the help of the referees who supplied us with excellent overviews and appraisals. In fact, this section contains literal quotes from the referee reports, and we would like to think of our contribution as mostly redactorial, trying to ensure a common format and uniform criteria of classification. In order to maintain the anonymity of the individual referees, we do not attribute the quotes we make. A list of all referees involved in the edition of this volume is included separately. We conclude in section 4 with a summary of some of the lessons that we have learned from this case study.

This article does not attempt to provide an in-depth analysis of the solutions to the case study: because we have contributed solutions ourselves it would have been difficult to ensure a truly impartial assessment of the other contributions. Besides, such an analysis would have taken more time and energy than

* This work was partially sponsored by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen"

we were prepared to expend. Nevertheless, we hope that our classification can be helpful to researchers and even practicing software engineers who try to apply formal methods to concrete problems, and can perhaps even indicate links between formal methods developed from different theoretical backgrounds. On the other hand, we are aware that our subjective backgrounds and predilections have influenced the presentation of the contributions in this overview. Any misrepresentations are entirely our fault.

2 The problem

This section reviews the specification problem, reproduced on pages 1–4 of this volume, and highlights key issues that the contributions were expected to address.

The problem calls for the specification of a series of components. The problem statement begins with a description of the procedure interface employed by all components. Each component is required to accept concurrent calls from different client processes, although there can be at most one outstanding call per process to facilitate identification of return values. Specification methods that emphasize modularity could be expected to give a separate specification of the interface behavior and reuse that specification in subsequent component specifications.

Next, the problem statement describes the individual components and poses five specific problems:

1. The first problem calls for the specification of a memory component that accepts Read and Write calls with appropriate parameters. The problem requires the memory to behave as if it consisted of an array of memory cells for which atomic read and write operations are defined. These atomic operations may fail and may be retried by the memory, hence a call to the memory may lead to the execution of several atomic read and write operations. Alternatively, the memory may raise the exception MemFailure, indicating that the operation may not have succeeded. (An unsuccessful Write operation may still have caused a successful atomic write to the memory.) Most contributors understood the wording of the informal description as a description of the externally visible memory behavior (as suggested by the phrase "as if"), not the actual memory operation. Specifically, the informal description explicitly stated the possibility of retries only for Write calls: whereas an external observer may be able to tell that a Write call has ben retried if there are concurrent Read calls for the same location, an observer will only be able to witness the effect of the last atomic read in response to a Read call. This issue has nevertheless caused some controversy, because the memory implementation (problem 3) describes an explicit mechanism for retries that allows several atomic reads in response to a Read call of the memory component. The solutions by Best, Romijn, and Hooman include a formal proof of the (observational) equivalence of the "single-atomic-read" and the

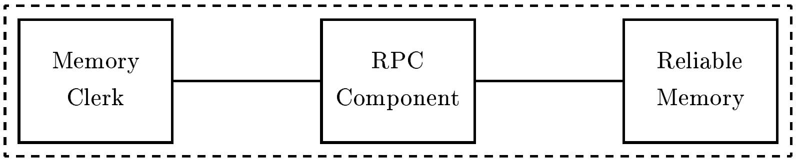

Fig. 1. Implementation of the memory component

"multiple-atomic-read" behaviors, whereas Larsen, Steffen, and Weise consider the problem statement to be flawed in this respect.

The first problem also introduces two variants of the memory component:

- the reliable memory component, which never returns a failure exception
- the ever-failing memory component, which always fails.

Participants were asked whether these variants are a valid implementation of the original memory component, and, if so, why this is a reasonable assumption. The solutions are unanimous in their answers: all regard both variants as a valid implementation. The rationale to consider the ever-failing memory an (unavoidable) implementation is that the specification cannot rule out a catastrophic failure of the memory. Several authors remark that probabilistic approaches could be employed to distinguish a memory that never works from one that fails only temporarily.

2. The second problem requires the specification of an RPC (remote procedure call) component. It offers a single RemoteCall procedure whose arguments are a procedure name and a list of arguments to be passed on to a server. If these arguments are "syntactically" correct, the RemoteCall is translated to an appropriate call of the server. When the server replies, the result is passed back to the client. However, there may also be failures, in which case the RPC component issues an RPCFailure exception.
3. The third problem asks for a formal proof of implementation of a memory component by the configuration of components shown in figure 1: A reliable memory is combined with an RPC component. The memory clerk component (that is only implicitly described in the problem statement) translates Read and Write calls to appropriate calls of the RPC's RemoteCall procedure and translates RPCFailure exceptions to MemFailure exceptions, ensuring that the implementation has the same interface as the memory component. The clerk may also retry calls for which the RPC has signalled an RPCFailure exception.

   It was assumed that any formalism designed for the specification of reactive systems would support the implementation of a single component by three components as indicated in figure 1. An important criterion was whether a rigorous, possibly computer-assisted, proof could be provided in the chosen framework.
4. The fourth problem describes a "lossy RPC" component, whose functional behavior is similar to the RPC component, but that guarantees a certain timing behavior. Specifically, it will both forward client calls to the server

and transmit results back to the client within δ seconds from when it has received the call or result. If the server does not respond, the lossy RPC component may fail to return a result to the client.

5. Finally, the fifth problem asks to prove that the original RPC component is implemented by a lossy RPC component and an auxiliary component (called the RPC clerk in most solutions) that raises an RPCFailure exception when more than $2\delta + \epsilon$ seconds have elapsed since the RemoteCall was received from the client, assuming that the server always responds within ϵ seconds. This assumption is necessary because the RPC implementation is allowed to return both an exceptional and a normal result if the server is too slow. Abadi, Lamport, and Merz explain that the informal description of the lossy RPC component and its clerk is problematic because it allows situations where a client process sends a second request before the lossy RPC has answered the first one. They suggest that it would have been sensible to replace the handshaking protocol imposed by the procedure interface by a protocol where a sender process can issue a new call after a certain timeout period. Cuellar, Barnard, and Huber introduce a timeout action that causes the lossy RPC component to forget a pending call.

3 The solutions

3.1 Criteria for classification

In tables 1 to 3 we have classified the solutions to the RPC memory specification problem according to a number of criteria that we explain now. Thereafter we give a short review for each solution.

Coverage This entry describes which subproblems have been addressed in the solution.

Means of presentation Most formalisms rely on a textual presentation of specifications as programs or formulas, sometimes even in machine-readable formats. Some formalisms additionally or exclusively provide a graphical notation for specifications, which may have a formal semantics or be purely illustrative.

Modularity We call a specification modular if it is subdivided into meaningful parts that can be understood independently separately and reused in different contexts.

Properties vs. operational We indicate whether the formalism emphasizes abstract, property-oriented descriptions of the interface or models specifications more operationally, for example using a programming notation. Typically, logic-based formalisms would fall under the first category, and process algebra or transition systems would represent the second one. In fact, the distinction is seldom clear-cut: firstly, many formalisms support both forms of specification, secondly, even formalisms that are more oriented towards operational specifications often require sophisticated liveness or fairness constraints. We indicate which specification style has been used in the solution of the RPC memory problem.

Solution: authors (number)	Coverage	Means of presentation	Modularity	Properties vs. operational
Abadi, Lamport, Merz (1)	1–5	TLA formulas, diagrams	yes	operational
Astesiano, Reggio (2)	1–3	structured nat. lang., algebraic spec.	yes	rather operational
Best (3)	1–5	annotated (timed) Petri nets	yes	operational
Blom, Jonsson (4)	1–3 (no proofs)	LTL formulas, diagrams	yes	mainly operational
Broy (5)	1–5 (no proof for 5)	predicates on streams	yes	property-oriented
Cuellar, Barnard, Huber (6)	1–5	Unity-like notation, temporal logic	yes	operational
Gotzhein (7)	1, short discussion of 2,3	temporal logic formulas	little (spec. of architecture)	property-oriented
Hooman (8)	1–5	PVS theories	yes	property-oriented
Hungar (9)	1–3 (incomplete specification)	timing diagrams, TL formulas, CSP	little	property-oriented
Klarlund, Nielsen, Sunesen (10)	1–3	logic on strings, Fido system descriptions	no	property-oriented
Kurki-Suonio (11)	1–5	TLA formulas, diagrams	yes	operational
Larsen, Steffen, Weise (12)	1–5	transition systems, diagrams	little (spec. patterns)	mainly operational
Romijn (13)	1–5	I/O automata	yes	operational
Stølen (14)	1–5	predicates on streams	yes	property-oriented
Udink, Kok (15)	1–3	Unity-like programs	yes	operational

Table 1. Classification of solutions

Hiding Formalisms that (mostly) rely on building an operational model of the specified system require a way to hide state components that are only introduced as auxiliary constructs to describe the model in order to avoid overspecification. This category indicates the presence of a hiding operator in the specification language. It does not apply to property-oriented specification formats.

Environment vs. component Some formalisms advocate a separation of environment and component specifications, for example to obtain an open-system specification. Other formalisms specify the overall behavior of the system together with its environment.

Stepwise refinement A few solutions indicate how the implementations described in the problem statement could have been derived from the abstract specification in a succession of refinement steps, sometimes presented as a succession of classes in an object-oriented development style. Although step-

Solution: authors (number)	Hiding	Environment vs. component	Stepwise refinement	Decomposition
Abadi, Lamport, Merz (1)	existential quantification	separate	no	yes (but not finite-state)
Astesiano, Reggio (2)	no (but "implementation function")	mixed	yes	no
Best (3)	restriction operator	no environment assumptions	yes	no
Blom, Jonsson (4)	existential quantification	mixed	no	yes (but not finite-state)
Broy (5)	not applicable	separate (ass.-commitment style)	partly	no
Cuellar, Barnard, Huber (6)	visibility annotations	separate	no	yes (finite state)
Gotzhein (7)	not applicable	mixed	only high-level specification	no
Hooman (8)	not applicable	separate	no	no
Hungar (9)	not applicable	separate (ass.-commitment style)	no	yes
Klarlund, Nielsen, Sunesen (10)	"hand-coded" in the proof	mixed	no	yes
Kurki-Suonio (11)	no	no	object-oriented reuse	no
Larsen, Steffen, Weise (12)	restriction operator	mixed	specification patterns	yes (finite-state)
Romijn (13)	external vs. internal actions	syntactic distinction	no	yes (but not finite-state)
Stølen (14)	yes (existentially quantified oracles)	input, output streams	yes	no
Udink, Kok (15)	local variables	mixed	yes	no

Table 2. Classification of solutions (continued)

wise refinement was not required in the RPC memory case study, we include this feature in the table.

Decomposition We indicate whether the component specifications have been further decomposed into several "lightweight" and largely independent processes. For example, the memory can be described as an array of memory cells that function uniformly. Similarly, all client processes may be treated in the same way by the components. Decomposition may help in the refinement proofs; it is a necessary prerequisite for those approaches that rely on model-checking a finite-state abstraction of the system.

Style of proof Proofs can be presented in various degrees of formalization. Informal, textbook-style proofs typically contain arguments about the operational behavior of a system. More rigorous proofs rely on "mathematical" (semantic) style of reasoning or are performed in a system of formal logic. Finally, in the case of machine-checked proofs, we distinguish between the

Solution: authors (number)	Style of proof	Related solutions
Abadi, Lamport, Merz (1)	formal	4, 11, 13, 6
Astesiano, Reggio (2)	outline	
Best (3)	informal	
Blom, Jonsson (4)	no proofs	1, 13
Broy (5)	mathematical	14, 10
Cuellar, Barnard, Huber (6)	model-checking	12, 9, 1
Gotzhein (7)	no proofs	
Hooman (8)	interactive	
Hungar (9)	model-checking	6, 12, 7
Klarlund, Nielsen, Sunesen (10)	decision procedure	6, 9, 12
Kurki-Suonio (11)	informal outline	1, 15
Larsen, Steffen, Weise (12)	model-checking, formal abstraction	6, 10, 3
Romijn (13)	mathematical	1
Stølen (14)	mathematical	5
Udink, Kok (15)	outline	6, 11

Table 3. Classification of solutions (end)

use of automatic decision procedures such as model checking and interactive theorem provers. Because all components are infinite-state systems, model checking can only be applied after building finite-state abstractions of the components.

Related solutions For each solution we indicate what we believe to be the most closely related solutions.

3.2 Reviews of individual contributions

We list the individual solutions in alphabetical order. For each solution we give a short summary and indicate what we believe are the most relevant issues raised in the contributions. These sections are largely based on the referees' comments. We are grateful for the permission to reprint excerpts from their evaluations.

Contribution 1 M. Abadi, L. Lamport, S. Merz: A TLA solution
The contribution presents a complete solution to the RPC-Memory specification problem. The specifications are presented as modules in the specification language TLA$^+$, which is based on the Temporal Logic of Actions (TLA). The concepts of TLA and TLA$^+$ required to understand the solution are explained as they are used. Several aspects of the solution are illustrated with the help of predicate-action diagrams, a graphical formalism whose semantics is given in terms of TLA formulas. The modules of TLA$^+$ allow for reuse of specifications, for example of the procedure interface specification or of basic "data types" such as sequences.

The logic TLA supports an operational style of specification, but also provides temporal logic operators to express standard invariant and liveness properties. Syntactic restrictions on well-formed TLA formulas ensure that all formulas are invariant under stuttering, hence the implementation relation can be expressed as logical implication. Existential quantification over state variables corresponds to hiding of internal state components. The authors give separate specifications of the environment and the components and indicate how one would write assumption-guarantee specifications. A non-interleaving style of specification helps to decompose the specifications into pieces of manageable complexity, which simplifies the verification tasks. The proofs are formal, based on the logical rules of TLA. Refinement mappings and history variables are used in the proofs of existentially quantified formulas. Proof outlines appear in the paper, the complete proofs are available separately. No machine assistance has been used in verification.

Related solutions include the DisCo approach presented in solution 11 by R. Kurki-Suonio, whose semantics is based on TLA. The TLT formalism of solution 6 by J. Cuellar, D. Barnard, and M. Huber has some similarity with TLA. Solution 4 by J. Blom and B. Jonsson, although based on standard linear-time temporal logic, uses a very similar structure of the specification, as does solution 13 by J. Romijn, which is based on I/O automata.

Contribution 2 E. Astesiano, G. Reggio: A Dynamic Specification of the RPC-Memory problem

The contribution covers problems 1 to 3 of the RPC-Memory problem. The main interest of the authors has been to suggest a methodology for the transition from a natural language description to a formal specification, using an intermediate, structured, informal specification. In particular, the component to be developed has to be delimited from its environment. The formal specifications are expressed as a combination of algebraic specifications, labelled transition systems (also presented in an algebraic way), and formulas of a CTL-like branching time temporal logic. The composition of separate components does not rely on predefined operators but is also specified explicitly in an algebraic style. Every specification is accompanied by an informal part written in structured natural language that explains the formal specification, discusses its requirements, and lists “shadow spots”, which represent inconsistencies or ambiguities in the informal document.

Refinement steps may produce so-called design specifications that contain only Horn clauses (and, in particular, no temporal logic formulas) and can be simulated using a rapid-prototyping tool. The implementation of the memory component by a system composed of a reliable memory, an RPC component, and a memory clerk is presented as a single refinement step. The paper contains an outline of an implementation proof, the full proof is available as a technical report.

The paper is unique in this volume in its emphasis on making the transition from informal to formal specifications an explicit part of formal system development. The authors emphasize the importance of validating an implementation with the help of rapid prototyping. On the other hand, formal verification appears to be less of a concern to the authors.

The formal basis of the approach is comparable to other contributions based on transition systems such as contributions 13 by Romijn or 12 by Steffen, Larsen, and Weise.

Contribution 3 E. Best: A Memory Module Specification using Composable High-level Petri Nets
This contribution covers all subproblems of the RPC-Memory specification problem. Specifications are expressed in a formalism called M-nets that combines annonated, high-level Petri nets with CCS-like composition operators such as parallel composition, restriction, and synchronous events. A real-time extension of M-nets is used for subproblems 4 and 5. The formalism and its use are explained in the paper. The implementation relation is defined as the inclusion of observable traces. The use of a common formalism for both specifications and implementations facilitates stepwise refinement as is illustrated in the paper: the memory specification is obtained by a series of refinements that either add requirements or allow additional behavior.

The formalism is largely based on graphical notation and employs operational concepts that are rather intuitive. Decomposition into concurrent subsystems is very natural with Petri nets, it also gives rise to an independence relation that aids in proving an implementation correct. Annotations at places and transitions are used to formulate constraints that cannot be conveniently expressed as a Petri net. The paper indicates the verification conditions necessary to prove the implementation relation. Textbook-style, informal refinement proofs are given in the appendix.

Contribution 4 J. Blom, B. Jonsson: Constraint oriented temporal logic specification
The contribution presents solutions to problems 1 to 3 of the RPC memory case study, excluding proofs. The authors use standard temporal logic of linear time (LTL) to specify the memory as a conjunction of constraints, each of which captures some aspect of the component's behavior and is of manageable complexity. This style of decomposition is facilitated by a non-interleaving model of the system, at the cost of introducing some non-obvious fairness requirements. The structure of the specification suggests a particular system architecture where different components interact via synchronous events (similar to process algebra) represented as flexible predicates. Each constraint is specified in a largely operational style, essentially as a transition system. Hiding of internal state components is achieved by flexible quantification of flexible variables. Parts of the specifications are explained with the help of diagrams that provide a graphical syntax for LTL formulas. The paper contains an outline of the topmost level of a refinement proof, but does not discuss its details.

Both the structure of the specification and the formalism are quite similar to contribution 1 by Abadi, Lamport, and Merz. Solution 13 by Romijn, which is based on I/O automata instead of temporal logic, uses a similar structure.

Contribution 5 M. Broy: A Functional Solution to the RPC-Memory Specification Problem
The contribution presents assumption-commitment specifications of all subproblems in the RPC-Memory case study. The specifications use a "black-box" perspective, describing the allowed input-output behavior of each component as a relation on timed streams with the internal structure of components hidden. It is shown that a complex specification can be made understood with the help of appropriately defined auxiliary symbols and predicates: each informal requirement of the specification problem is matched with a clause in the formal, stream-based specification. The paper contains a short introduction to the formalism, a more detailed exposé is given in the appendix together with an alternative, trace-based specification of the memory component.

The contribution emphasizes modularity of specifications: each requirement can be understood independently from the others. This is possible because the specification expresses a relation between streams (i.e., values) rather than variables or events (i.e., names). Modularity helps in verifying that the implementation is a refinement of the high-level specification. The necessary proofs concerning problem 3 of the case study are given in the appendix.

The most closely related contribution is that by Stølen (14). The contribution 10 by Klarlund, Nielsen, and Sunesen is also based on a stream model, but its internal structure is quite different, motivated by the use of a formal (monadic second-order) logic and the emphasis of associated decision procedures.

Contribution 6 J. Cuellar, D. Barnard, M. Huber: A Solution relying on the Model Checking of Boolean Transition Systems
This contribution addresses all parts of the RPC-Memory case study. Specifications are expressed in the language TLT (temporal logic of transitions), which combines a Unity-like programming notation with a temporal logic that bears some resemblance to TLA. The formalism supports communication via shared variables as well as synchronous communication, modelled as joint actions of the environment and the component. Visibility annotations in variable declarations serve to distinguish between state components of the environment and the component. Assumptions about the behavior of the environment may be stated in a special section of TLT modules; they give rise to verification conditions when modules are composed. The module system helps to break specifications into manageable pieces, but has a rather complicated, non-compositional semantics.

The main emphasis in this solution is on abstraction and decomposition. It is shown how a factorization technique can be used to reduce the implementation proof to a finite-state problem, which has reportedly been handled by the TLT model checker. However, the paper is a little vague about the details of the actual verification. It is not clear whether the real-time implementation proof has been handled by the model checker or not.

Contribution 12 by Steffen, Larsen, Weise and contributionand 9 by Hungar rely on similar abstraction techniques in the context of modal transition systems and branching-time temporal logic. Also related is the paper 10 by Klarlund, Nielsen, and Sunesen in its use of automatic decision procedures in verification.

Udink and Kok (solution 15) base their contribution on a somewhat similar variant of Unity, while the temporal logics used in solutions 1 by Abadi, Lamport, Merz and 4 by Blom and Jonsson are related to the temporal logic part of TLT.

Contribution 7 R. Gotzhein: Applying a temporal logic
The paper presents a solution to problem 1 of the RPC-Memory case study and discusses how the implementation of the memory (problems 2 and 3) could be described and verified. The author advocates an approach where behavior and system architecture are specified together and demonstrates how such a formalism could be defined. The architecture is given as a network of agents, connected through interaction points. The approach involves a notion of refinement that allows both agents and interaction points to be refined into more detailed networks. The functional specification of an agent is expressed as a property-oriented description of its behavior at the interface level, expressed as a list of temporal logic formulas.

Rather than giving a fixed temporal logic, the author suggests to choose a logic that is sufficiently expressive for the specification problem at hand. For the RPC-Memory problem he chooses a many-sorted first-order branching time logic with operators to refer to the future, to the past, to actions, to the number of actions, and to intervals. The specification is expressed as one page of formulas of this logic. The semantic definitions necessary to understand the specification are given in the paper.

Because it does not define the memory implementation, the paper does not include any proofs, although refinement is discussed at a general level. The expressiveness of the logic leaves some doubts whether formal verification would actually be feasible. The memory specification is already of worrisome complexity, maybe indicating that a purely behavioral approach is not very well suited for this kind of specification problem.

Contribution 8 J. Hooman: Using PVS for an Assertional Verification of the RPC-Memory Specification Problem
This contribution presents a complete solution of the RPC-Memory problem carried out in an assertional framework developed over several years by the author. All specifications and verification conditions have been checked mechanically with the help of the interactive theorem prover PVS (Prototype Verification System). The author uses the higher-order logic of PVS to express properties that are more conventionally expressed in some kind of temporal logic. The framework and its support by PVS are explained in the article.

In this approach, specifications are expressed as assumption-commitment pairs that define a formal theory of the component's behavior in a well-behaved environment. For the RPC-Memory problem, however, environment and component are modelled as a single system, hence all environment assumptions are trivial. The formalism is property-oriented, constraining the acceptable behaviors (sequences of events) at the interface level. The contribution shows that machine-checked (although not automatic) verification of the RPC-Memory system is possible even without decomposition of the component specifications into "lightweight" processes.

The contribution is unique both in the theoretical underpinnings of the assertional approach it uses and in the use of an interactive theorem prover as a tool to write specifications and prove theorems about them.

Contribution 9 H. Hungar: Specification and Verification Using a Visual Formalism on Top of Temporal Logic
This contribution covers problems 1 to 3 of the RPC-Memory case study. Technically, specifications are expressed in an assumption-commitment variant of the branching-time temporal logic CTL, using a property-oriented specification style. The specifications are expressed with the help of symbolic timing diagrams (STDs), whose formal semantics is given by a translation to CTL. However, with some practice, STDs can be understood intuitively. Both the logic and the diagrams are introduced in the paper to the extent necessary to be able to read the specifications. The author has, in his own words, mainly been concerned with "the formal verification of key properties of a design, [...] less emphasis [has been] put on issues like completeness of the specification". In fact, he gives an example of a run that satisfies the specification, but does not conform to the informal memory specification. He argues that such behaviors are too pathological to occur in any realistic implementation.

The paper also includes CSP-like programs for the memory, RPC, and memory clerk components and explains techniques such as decomposition, abstraction, and model checking to verify such programs against CTL specifications. Part of the verifications required for the RPC-Memory problem has reportedly been performed with the help of automatic tools developed at the University of Oldenburg, although no details of the verification are given in the paper.

The use of abstraction and model-checking is similar to the frameworks used in contribution 6 by Cuellar, Barnard, and Huber and contribution 12 by Larsen, Steffen, and Weise. Contribution 7 by Gotzhein is another solution that relies on property-oriented temporal-logic specifications.

Contribution 10 N. Klarlund, M. Nielsen, K. Sunesen: A Case Study in Verification Based on Trace Abstractions
The paper addresses problems 1 to 3 of the RPC-Memory specification problem using behavioral component specifications expressed in a monadic second-order logic over finite strings, which allows to specify the safety part of the specifications. The system description language Fido provides a high-level notation that can be translated into formulas of the target logic. The verification part is carried out with the help of an automatic decision procedure called MONA.

The emphasis in this solution is on the availability of an automatic decision procedure for the considered logic. In particular, all theorems have been checked by the MONA tool. However, the correctness of the underlying abstractions to finite-state systems is left implicit in the paper (strictly speaking, only a finite-state instance of the problem has actually been verified). The implementation relation is defined as inclusion of observable traces, where the notion of observability is defined by the user as appropriate for the problem at hand. For the present case study, the authors notice that atomic reads should not be made observable unless a Read call to the memory may induce several atomic reads.

The paper is related to the solutions 6 by Cuellar, Barnard, and Huber, 9 by Hungar, and 12 by Larsen, Steffen, and Weise in its use of finite-state abstractions to perform automatic verification. The semantic model is related to the stream model used in the solutions 5 by Broy and 14 by Stølen, but is restricted to safety properties.

Contribution 11 R. Kurki-Suonio: Incremental Specification with Joint Actions: The RPC-Memory Specification Problem
The contribution covers all aspects of the RPC-Memory case study. The solution is developed in an object-oriented fashion that emphasizes the reuse of subcomponents and basic patterns of interactions. The computational model is based on the synchronization of environment and component via joint actions. The semantic basis for the formalism is Lamport's Temporal Logic of Actions (TLA), with some additional ideas from Back's refinement calculus. The method is supported by prototyping tools that animate specifications. However, the aspect of tool support is not discussed in this paper.

The specifications are presented as a succession of class and action definitions, presented as TLA formulas. Several diagrams, including state charts, provide additional explanation, although they are not given a semantics of their own. The emphasis in this approach is on modularity: Certain restrictions on the allowed modifications of an action in a subclass ensure that safety properties of a class are inherited by subclasses, at least for the untimed part. Each class definition can be understood as defining an automaton that is further refined in subclasses. The lack of distinction between external and internal state components and between environment assumptions and component commitments is justified by the emphasis on producing models rather than abstract specifications. The method does however provide the vehicle of "ghost variables" that allows state variables of a superclass to be eliminated in a subclass when they are no longer needed. The approach favors the development of an implementation by stepwise refinement where correctness (at least for safety properties) is guaranteed by construction. For the present case study, the implementation consists of components that have been developed separately, so there is a need for separate correctness proofs. The paper includes informal arguments why the implementation is correct and shows how these could be turned into formal TLA proofs of implementation.

Solution 1 by Abadi, Lamport, and Merz is related in its use of TLA as its semantic basis. The specification language contains ideas similar to those underlying Unity and TLT (see contributions 15 by Udink and Kok and 6 by Cuellar, Barnard, and Huber).

Contribution 12 K. Larsen, B. Steffen, C. Weise: The methodology of modal constraints
The contribution addresses all subproblems of the case study, except that no liveness properties are specified. The specifications are written as modal transition systems, which add a distinction between "may" and "must" transitions and a form of conjunction to constructs from process algebras such as CCS, resulting in an interesting mix of operational and property-oriented specification

styles. The underlying theory is explained to the extent necessary to understand the solution.

The individual specifications are expressed in the form of transition diagrams that reuse a number of basic patterns. The specification is decomposed into fine-grained specifications that concern individual memory locations, client processes, and memory values. This decomposition together with an abstraction step allows for the use of a model checker for verification, which has reportedly been performed using the TAV tool developed by the authors. The price to pay is a highly complex structure of the specification. It was not clear to the referees whether the specification actually conforms to the informal description or not. In particular, modelling the dependency between write and read actions turned out to be non-trivial and resulted in a complex transition system.

The paper is related to solution 6 by Cuellar, Barnard, and Huber, solution 9 by Hungar, and solution 10 by Klarlund, Nielsen, and Sunesen in its use of finite-state abstractions and model checking.

Contribution 13 J. ROMIJN: TACKLING THE RPC-MEMORY SPECIFICATION PROBLEM WITH I/O AUTOMATA
Romijn's contribution gives solutions for all subproblems of the RPC-Memory case study. The specifications are expressed in the framework of I/O automata with fairness constraints. The timed part of the specification problem required an extension of the model of fair I/O automata. The distinction between internal, external, and environment events is built into the semantics of I/O automata, although it is noted in the paper that this "syntactic" distinction is not enough, for example, to express the assumption that certain environment actions happen within a specified time bound.

Each component specification is given as the composition of one I/O automaton per client process. The paper includes a statement of all verification conditions necessary to prove refinement and sketches their proofs, including a proof that a memory component that allows multiple atomic reads per Read request cannot be distinguished from one that allows at most one atomic read. An addendum to the contribution, which is available separately, contains the complete proofs for all theorems in mathematical and partly formal (predicate logic) style.

The structure of the specifications is similar to that of contributions 1 by Abadi, Lamport, and Merz and 4 by Blom and Jonsson, although the semantic basis is different.

Contribution 14 K. STØLEN: USING RELATIONS ON STREAMS TO SOLVE THE RPC-MEMORY SPECIFICATION PROBLEM
The paper addresses all subproblems of the RPC-Memory case study. Component specifications are expressed as relations between input and output streams, which model communication histories of input and output channels. Composition of specifications corresponds to the conjunction of the input-output relations. A specification refines another specification if any input-output behavior of the former is also an input-output behavior of the latter. Thus, refinement corresponds

to logical implication. The paper distinguishes between time-independent and time-dependent specifications. A time-independent specification is based on untimed streams. A time-dependent specification employs timed streams and can express timing constraints and causalities.

The paper deviates from the problem statement in that the handshake protocol is not imposed. This means for example that the user may send a new call before the memory component issues a reply to the previous call by the same user. An appendix shows how handshake communication can be introduced as a refinement.

The specifications are developed in several steps, starting from an unfailing memory for one client process. The paper emphasizes the use of oracles to describe the time-independent nondeterministic behavior in a structured way. Essentially, an oracle represents the nondeterministic choices made by the component; it can be viewed as an additional, hidden input stream to the specification describing the functional component behavior. Constraints on oracles impose fairness or compatibility requirements. The paper gives conventional mathematical proofs for the correctness of the implementation.

The contribution is related to solution 5 by Broy.

Contribution 15 R. Udink, J. N. Kok: The RPC-Memory Specification Problem: Unity + refinement calculus
The contribution covers subproblems 1 to 3 of the RPC-Memory case study. The authors have extended the Unity specification language with a module system that provides local and global variables. On the logical level, they have added concepts from the refinement calculus developed by Back and Kurki-Suonio. The individual specifications are given as modules in this Unity-like language.

The specification style is operational, internal and external variables are distinguished by the module system. The method provides transformation rules that preserve all temporal properties of programs, even when applied to a module in isolation. The paper contains an outline of the necessary steps in the refinement proof for the memory component. The method is currently not supported by interactive or automatic proof checkers.

The contribution is related to the solution 6 by Cuellar, Barnard, and Huber in its use of a Unity-like framework, and to solution 11 by Kurki-Suonio in its emphasis on transformational derivation and its foundation in the action systems formalism.

4 Conclusion

Although we do not want to and cannot come to a final judgement, it is helpful to draw some conclusions. Which of the contributions as solutions to the RPC case study a reader may prefer, depends very much on taste and style. The idea of the whole experiment was never to have Olympic Games in specification, refinement, and verification. However, it is worthwhile to draw some final conclusions.

First of all, it was very interesting to see how the choice of the individual methods that were used to tackle the problem was often less important for the quality of the final solutions than the modeling ideas of the specifiers. Certainly, the creativity and expertise, and of course also the routine of the person applying a method, are more important for the result than the particular notation and model used. Of course, there are methods and models which do not provide certain helpful concepts and, therefore, make the life of the specifier a little harder. However, even plain predicate logic is a very powerful tool, and if a method comprises first-order predicate logic or at least a sufficient fragment of it, many ideas can be expressed, maybe less explicitly, but nevertheless they can be made to work. This is certainly not surprising. It is like it is for programming languages: a good programmer can write good quality programs even in a bad programming language, and a bad programmer can write bad programs even in a good programming language.

As a consequence of the observations we made above, it was interesting to see how over the time working on the case study the authors of the contributions were concentrating more and more on aspects of how to apply their methods, such that theoretical questions became less and less important. This was certainly what we expected and what we intended with the case study. The idea was to stimulate researchers in that area to concentrate more on application issues and less on maybe not so important theoretical aspects. This worked out perfectly.

Another thing which worked out very convincingly was the cross-refereeing process. It showed that a very fruitful discussion was possible between proponents of quite different methods. Also here, the case study was a major contribution to improving the understanding of researchers working in the field of different methods. And this is, finally, what we would really like to achieve: to make sure that researchers in that area look over the fence, respect the advantages of other approaches, learn enough about them to be able to combine them in such a way that it finally leads to a comprehensive understanding of the field and of how the methods can work together, such that we come step by step closer to a practical application.

An aspect, very important for practical considerations, is the economy of a method. How long does it take to learn it, how long does it take to use it for a particular problem. Here our little experiment does not provide much input. We think most of the methods around and also those used in this volume are not at a stage where these questions can be tackled successfully.

Acknowledgements We gratefully acknowledge helpful comments by M. Abadi, E. Best, L. Lamport, and K. Stølen on previous versions of this paper.