Specification Problem

Manfred Broy[1] and Leslie Lamport[2]
[1]Institut für Informatik, Technische Universität München
[2]Systems Research Center, Digital Equipment Corporation
Email: broy@informatik.tu-muenchen.de, lamport@src.dec.com

1 The Procedure Interface

The problem calls for the specification and verification of a series of *components*. Components interact with one another using a procedure-calling interface. One component issues a *call* to another, and the second component responds by issuing a *return*. A call is an indivisible (atomic) action that communicates a procedure name and a list of *arguments* to the called component. A return is an atomic action issued in response to a call. There are two kinds of returns, *normal* and *exceptional*. A normal call returns a *value* (which could be a list). An exceptional return also returns a value, usually indicating some error condition. An exceptional return of a value e is called *raising exception* e. A return is issued only in response to a call. There may be "syntactic" restrictions on the types of arguments and return values.

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

2 A Memory Component

The component to be specified is a memory that maintains the contents of a set MemLocs of locations. The contents of a location is an element of a set MemVals. This component has two procedures, described informally below. Note that being an element of MemLocs or MemVals is a "semantic" restriction, and cannot be imposed solely by syntactic restrictions on the types of arguments.

Name	Read
Arguments	loc : an element of MemLocs
Return Value	an element of MemVals
Exceptions	BadArg : argument loc is not an element of MemLocs. MemFailure : the memory cannot be read.

Description	Returns the value stored in address loc.

Name	Write
Arguments	loc : an element of MemLocs val : an element of MemVals
Return Value	some fixed value
Exceptions	BadArg : argument loc is not an element of MemLocs, or argument val is not an element of MemVals. MemFailure : the write *might* not have succeeded.
Description	Stores the value val in address loc.

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

Define an *operation* to consist of a procedure call and the corresponding return. The operation is said to be *successful* iff it has a normal (nonexceptional) return. The memory behaves as if it maintains an array of atomically read and written locations that initially all contain the value InitVal, such that:

- An operation that raises a BadArg exception has no effect on the memory.
- Each successful Read(l) operation performs a single atomic read to location l at some time between the call and return.
- Each successful Write(l, v) operation performs a sequence of one or more atomic writes of value v to location l at some time between the call and return.
- Each unsuccessful Write(l, v) operation performs a sequence of zero or more atomic writes of value v to location l at some time between the call and return.

A variant of the Memory Component is the Reliable Memory Component, in which no MemFailure exceptions can be raised.

Problem 1 (a) Write a formal specification of the Memory component and of the Reliable Memory component.

(b) Either prove that a Reliable Memory component is a correct implementation of a Memory component, or explain why it should not be.

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

3 Implementing the Memory

3.1 The RPC Component

The RPC component interfaces with two environment components, a *sender* and a *receiver*. It relays procedure calls from the sender to the receiver, and relays the return values back to the sender. Parameters of the component are a set Procs of procedure names and a mapping ArgNum, where ArgNum(p) is the number of arguments of each procedure p. The RPC component contains a single procedure:

Name	RemoteCall
Arguments	proc : name of a procedure
	args : list of arguments
Return Value	any value that can be returned by a call to proc
Exceptions	RPCFailure : the call failed
	BadCall : proc is not a valid name or args is not a syntactically correct list of arguments for proc.
	Raises any exception raised by a call to proc
Description	Calls procedure proc with arguments args

A call of RemoteCall(proc, args) causes the RPC component to do one of the following:

- Raise a BadCall exception if args is not a list of ArgNum(proc) arguments.
- Issue one call to procedure proc with arguments args, wait for the corresponding return (which the RPC component assumes will occur) and either (a) return the value (normal or exceptional) returned by that call, or (b) raise the RPCFailure exception.
- Issue no procedure call, and raise the RPCFailure exception.

The component accepts concurrent calls of RemoteCall from the sender, and can have multiple outstanding calls to the receiver.

Problem 2 Write a formal specification of the RPC component.

3.2 The Implementation

A Memory component is implemented by combining an RPC component with a Reliable Memory component as follows. A Read or Write call is forwarded to the Reliable Memory by issuing the appropriate call to the RPC component.

If this call returns without raising an RPCFailure exception, the value returned is returned to the caller. (An exceptional return causes an exception to be raised.) If the call raises an RPCFailure exception, then the implementation may either reissue the call to the RPC component or raise a MemFailure exception. The RPC call can be retried arbitrarily many times because of RPCFailure exceptions, but a return from the Read or Write call must eventually be issued.

Problem 3 Write a formal specification of the implementation, and prove that it correctly implements the specification of the Memory component of Problem 1.

4 Implementing the RPC Component

4.1 A Lossy RPC

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

- The RPCFailure exception is never raised. Instead of raising this exception, the RemoteCall procedure never returns.
- If a call to RemoteCall raises a BadCall exception, then that exception will be raised within δ seconds of the call.
- If a RemoteCall(p, a) call results in a call of procedure p, then that call of p will occur within δ seconds of the call of RemoteCall.
- If a RemoteCall(p, a) call returns other than by raising a BadCall exception, then that return will occur within δ seconds of the return from the call to procedure p.

Problem 4 Write a formal specification of the Lossy RPC component.

4.2 The RPC Implementation

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

Problem 5 (a) Write a formal specification of this implementation.
(b) Prove that, if every call to a procedure in Procs returns within ϵ seconds, then the implementation satisfies the specification of the RPC component in Problem 2.

Solutions

A TLA+ Solution to the “Dagstuhl Problem”

Martín Abadi[1], Leslie Lamport[1], and Stephan Merz[2]
[1] Systems Research Center, Digital Equipment Corporation
[2] Institut für Informatik, Technische Universität München
Email: {ma,lamport}@src.dec.com, merz@informatik.tu-muenchen.de

We give specifications for the unreliable and reliable memory components, the RPC component, and of clerks that connect these components in TLA+, as well as rigorous proofs of refinement between systems built from these components.

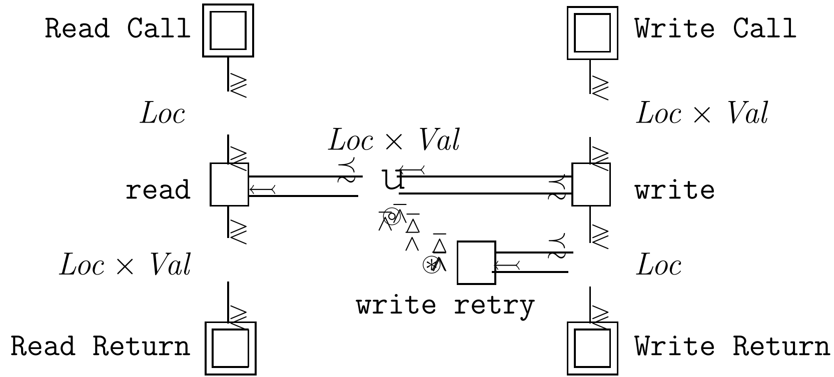
Our specifications are written as open system (“assumption/guarantee”) specifications, using an interleaving style with synchronous communication. Component behaviors are specified in terms of hidden (existentially quantified) internal state components. We reuse previously established specifications for the data part and the real-time aspects.

The refinement proof illustrates that a standard composition rule proved by the first two authors can be applied to a practical example of moderate size. Our proofs are rigorous rather than formal. We give a short discussion on what parts could be mechanized.

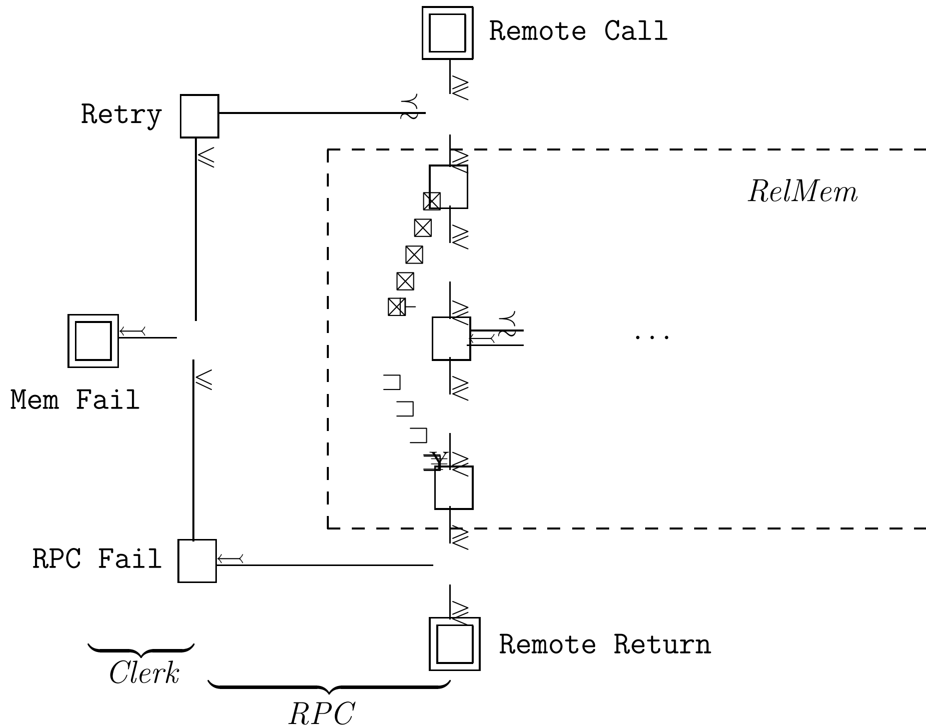
The Dagstuhl Problem by Broy and Lamport A Solution Based on Composing High Level Nets at Transition Interfaces

Eike Best
Institut für Informatik, Universität Hildesheim
Email: e.best@informatik.uni-hildesheim.de

A framework is presented in which the transitions (and the places, but this is of no importance for this problem) of a High Level Petri Net can have two kinds of annotations: one kind governs their vertical unfolding into elementary nets as usual, the other kind governs their horizontal composition. A reliable memory `RelMem` can be specified essentially in the following way (interface transitions are represented by double squares):



The picture is parametrised by p, the identity of the calling process. An unreliable memory `Mem` can be specified in a similar way by adding exception transitions to all places except the middle one which models the memory. `Mem` can be realised as $[Clerk \| RPC \| RelMem]$ synchronising at transitions in the following way (only for `Read`):



By a set of contractions (replacing a sequence of transitions by a single one) and other simple net transformations that preserve behaviour, $[Clerk \| RPC \| RelMem]$ can be shown to implement `Mem` as required.

A process algebraic approach to the Dagstuhl problem

Ed Brinksma
Faculteit der Informatica, Universiteit Twente
Email: brinksma@cs.utwente.nl

In our presentation we present the outline of a process algebraic solution of the workshop problem. In our approach we use process algebraic equivalences (laws) as design transformation principles to derive the desired implementation from the specification in a stepwise manner. In our proof we exploit the syntactic appearance (intension) of our expression as an important heuristic tool to guide this refinement.

A solution using stream processing functions

Manfred Broy
Institut für Informatik, Technische Universität München
Email: broy@informatik.tu-muenchen.de

We give a functional specification of the reliable and the unreliable memory component, of the rpc component and of the driver of the rpc component. The components are modeled by functions that map communication histories for the input channels onto communication histories for the output channels. A communication history for n channels is given by n streams of messages. Each stream contains an infinite number of time ticks. This allows to refer to the relative timing of messages on different channels.

We write the specification of the memory in an assumption/commitment style. The assumption is that every processor issues its next procedure call only after the previous one had returned. In the specification of the commitments the memory component we use a formula that specifies the relationship between the input stream and the output stream with the help of a internal access stream. It is required that every call returns and that the access stream and the calls and returns are in the correct relationship. Also for the RPC component an assumption/commitment style specification is given.

The three components can be composed. The composed system can be proved to fulfill the specification of the memory component again. In an analogous way the timed rpc component and the timed driver are defined and composed with the memory again. It can be proved that, if the memory responds fast enough, the component obtained by the composition fulfills the memory specification again.

The Dagstuhl problem – A Solution using TLT

Jorge Cuellar, Martin Huber, and Isolde Wildgruber
Siemens Corporate Research and Developement, ZFE BT SE11
Email: {jorge,mar,isolde}@zfe.siemens.de

TLT is a new language for writing specifications, abstract programs, implementations and their properties. Actually, all of them are just *global* TLT formulas. A program F satisfies a property Φ is thus expressed as $\mathsf{F} \Rightarrow \Phi$. A program F_1 is an implementation of the abstract program F_2 is just $\mathsf{F}_1 \Rightarrow \mathsf{F}_2$. The *local* properties (on states or predicates) are defined, as in μ-calculus, as (possibly) alternating fixed-points. TLT may be seen as a generalization of UNITY or as a branching time version of TLA. Compared to UNITY, TLT programs have more structure (for instance, local variables, local states and invisible internal actions). Programs are connected using link-constraints, interfaces and connectors. This enables communication through variables as well as transitions (actions). In TLT we have a fairness section in each program where instruction can be designated as WEAK or STRONG fair. Due to the guarded-command style of the language these fairness requirements can easily be formulated in the TLT logic and realized operationally in Kripke Structures as acceptance conditions. The Dagstuhl problem was fully described and proven with TLT.

A Solution with Process Algebra

Rob van Glabbeek
Department of Computer Science, Stanford University
Email: rvg@cs.stanford.edu

The components of the specification problem are represented by expressions in a CCS/ACP-like language. The real-time syntax is somewhat ad hoc, but can be translated in various real-time process algebras. All specifications of the problem together take only 2 slides. The statements about implementation are formalized by choosing a preorder between process expressions. Here I take two preorders in consideration: one based on failures/testing with a form of Kooman's Fair Abstraction Rule, and one slightly non-interleaving version of this preorder, based on a weaker concept of fairness developed with Frits Vaandrager in 1987 in the context of Petri nets. Proofs are not provided.

Applying a Temporal Logic to the "Specification Problem"

Reinhard Gotzhein
University of Kaiserslautern
Email: gotzhein@informatik.uni-kl.de

The unreliable and the reliable memory component 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 (i.e., active components) and interaction points (i.e., 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 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.

Assertional Specification using PVS

Jozef Hooman
Eindhoven University of Technology
Email: wsinjh@win.tue.nl

The "Specification Problem" is specified and verified in an assertional formalism, where properties of components are expressed by logical formulas. Design steps are verified in a compositional framework which allows reasoning with the specifications of components without knowing their implementation. Verification is supported by the interactive proof checker PVS (Prototype Verification System).

In the "Specification Problem" first properties of the memory component are specified in an untimed formalism, using ordered events. Next we show that the memory component is implemented by a reliable memory, an RPC, and a link component. To specify the lossy RPC component, the framework is extended with primitives to express the timing of events. This leads to an implementation of the RPC component. The tool PVS has been used to construct the correctness proofs of these implementations.

The Memory Component: Specification and Verification Using an MCTL-Based Method

Hardi Hungar
Kuratorium OFFIS, Oldenburg
Email: hungar@informatik.uni-oldenburg.de

The specification is done in Josko's branching time temporal logic MCTL. This logic extends CTL by allowing assumption/commitment formulas, which are very useful for the specification and verification of open systems, i.e. systems designed to work in a specific environment. Some of the specifications are given in a graphical formalism (as *Symbolic Timing Diagrams*) which often allows a concise representation of otherwise lengthy formulas.

The specification does not refer to the internals of the memory (its locations). Instead, the memory is specified directly in terms of its observable input/output behavior. Therefore, it allows a wide spectrum of implementations. Special care is taken also to specify in such a way that there is a finite state implementation of the control part of the memory, i.e. to guarantee the existence of an implementation.

Another important point is verifiability of an implementation against the specification. For MCTL, a verification methodology is available which combines model checking, compositional reasoning and abstraction techniques. The methodology relies to a large extent on automatic procedures and is supported in a prototype tool. An implementation of the memory component has been partially verified with the help of that tool. Note that restrictions in the expressive power of MCTL (i.e. no existentially quantified state variables) are important for a high degree of automation in the verification process.

Process structure and linear-time temporal logic

Bengt Jonsson
Department of Computer Systems, University Uppsala
Email: bengt@docs.uu.se

We present a specification of the memory interface, proposed by Broy and Lamport as a common case study for the Dagstuhl workshop in September 1994. The specification is carried out in first-order linear-time temporal logic, essentially in the restricted version of TLA (Temporal Logic of Actions) by Lamport. The main contribution of our specification is to demonstrate different structuring mechanisms that can be attained in LTL. For instance, we show how to emulate process-algebraic structuring techniques in LTL. Different parts of the specification represent different processes, which communicate through synchronizing actions. In this particular specification, the

different main components of the interface — the inner memory component, the remote procedure component, and the component that calls the remote procedure interface — are specified separately and combined using logical conjunction. Within a component — e.g., the inner memory interface — each call is represented as an "object". Specifications of the different objects are combined using universal quantification over object identifiers. We briefly outline how the structure of a verification could be achieved, although we do not carry out the actual verification.

Convenient Executions and Loosening: the Specification Problem

Shmuel Katz
Computer Science Department
The Technion, Haifa
Email: katz@cs.technion.ac.il

A version of the temporal logic $ISTL^{\wedge}$ is used to demonstrate a two-stage approach to specification and refinement of the specification problem presented by M. Broy and L. Lamport. In our approach, a partial order of events that correspond to a single execution is identified with the set of all linearizations of global states that are consistent with the partial order. Such a set is called an *interleaving set*, and each linearization is called an *execution sequence*.

In refinements, first convenient lower level executions are shown to implement execution sequences of upper level operations. The convenient executions at the lower level are precisely those where the lower level operations that implement a higher level one are all done sequentially, with no other lower level operations interspersed. These are legal lower level executions, even if they are unlikely to occur in practice because the operations are distributed in a collection of asynchronously executing processors. A mapping function from each convenient execution to some abstract one is generally simple and iterative.

Then in the second stage, all other computations are shown to be equivalent to one of the convenient ones. The equivalence maintains the ordering of all causally dependent events, but allows independent events to occur in different orders. This stage could be considered as a 'loosening' of the ordering imposed by the convenient executions. The advantage of this separation is that different kinds of reasoning and induction can be used for the two aspects. The convenient executions, the general computations, and the independence conditions are all expressed as assertions in the logic $ISTL^{\wedge}$.

In order to be faithful to the informal statement of the specification problem, the first part of the problem is expressed directly as convenient sequences. That is, the specification itself is that every computation is equivalent to one

of the convenient ones, as seen also in database serializability and sequential consistency of caching algorithms.

Specification and Refinement with Joint Actions: The Dagstuhl Example

Reino Kurki-Suonio
Tampere University of Technology
Email: rks@cs.tut.fi

Specifications for the "Dagstuhl Problem" are derived incrementally, using ideas of the DisCo language. The approach can be characterized by superposition-based transformations, object-oriented inheritance, and specialization of multi-object actions. Timing properties are introduced by two special-purpose variables (clock and current deadlines), one additional parameter in each action (moment of execution), and some strengthening of actions.

Formality: Joint actions can easily be mapped into TLA actions, and TLA is used for proofs. Derivation steps preserve safety properties, but liveness properties need separate proofs. However, implementations cannot always be derived directly from specifications, in which case correctness proofs are based on constructing "synchronized combinations" of specifications and implementations.

Informality: Tools for visualizing and animating (instances of) specifications encourage their informal inspection and validation, as well as formulation of properties that need to be proved.

Formally-Driven Semi-Formal Specifications

Egidio Astesiano and Gianna Reggio
Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Email: reggio@disi.unige.it

We present how an established formal specification technique for concurrent systems could be complemented by a method for developing specifications based on simple basic concepts, which may be also introduced in an informal way.

Thus following that method it is possible to develop specifications which are informal (natural language and graphical notations) and have associated a corresponding isomorphic formal specification.

This method is then applied to the proposed problem, giving two isomorphic specifications, one informal and one formal.

The Memory Component: Specifying Properties of Synchronized Causal Chains

Wolfgang Reisig
Institut für Informatik, Humboldt-Universität Berlin
Email: reisig@informatik.hu-berlin.de

A most elementary framework is employed, with dynamic predicates (sets) and actions that change membership (extension) of those predicates.

This approach, likewise taken in the Chemical Abstract Machine, the Γ-language and the evolving algebra concept, is equipped with a specific notion of runs (executions, computations) in the framework of high-level Petri Nets. Each action affects a limited set of dynamic predicates, hence several actions may occur independently. A single run then consists of a partial order of action occurrences.

A *line* in such a partial order is a maximal subset of totally ordered elements. Lines help to specify the memory component: The action occurrences affecting a memory location form a line. Each access of a process to the memory likewise forms a line. The set of *acceptable* runs is characterized this way.

Call a high-level Petri Net a *memory net* if it has two distinguished input places (*inread* and *inwrite*) and two distinguished output places (*outread* and *outwrite*). Likewise, an *environment net* has *outread* and *outwrite* as input places, and *inread* and *inwrite* as output places.

Compositions of a memory net with an environment net are gained by identifying equally labelled places.

The *specification* of *memcomp* is now gained easily: a memory net specifies *memcomp* if its composition with any environment yields a net that has only *acceptable* runs.

The RPC component can now be specified in the same style as *memcomp* described above, as a special kind of environment nets with new I/O places and with runs that are acceptable if they fulfill further requirements.

A Process Algebraic Approach Using Action Refinement

Arend Rensink
Institut für Informatik, Universität Hildesheim
Email: rensink@informatik.uni-hildesheim.de

We have tried to formulate a solution of the problem posed by Broy and Lamport on the basis of the principle of *action refinement*. As a basic formalism we have used process algebra, in particular the LOTOS notation; this forces

us to take into account aspects of synchronous communication and refusal of actions. The fact that we have used the standardised language LOTOS made it possible to give fully formal descriptions of the data aspects of the problem.

We started with a high-level specification of the memory cell, in which read and write actions are given as single, atomic actions. The intention was to show that the RPC-implementation refines this under a refinement function that maps the atomic actions onto the required RPC protocols. Unfortunately, this turned our to be problematic: the repetition of writes in the implementation makes interference patterns possible that do not correspond to any behaviour of the specification. For the read actions, on the other hand, the refinement idea carries through without any problems.

Our conclusion is that the concept of *atomicity* can be stretched to some degree (atomic read actions could be refined into larger protocols) but not arbitrarily far (atomic write actions could not be refined into multiple ones, since the latter generate multiple visible state changes).

Dagstuhl-Seminar Specification Problem – a Duration Calculus Solution

Marcin Engel[1] and Hans Rischel
Department of Computer Science
Technical University of Denmark
DK 2800 Lyngby, Denmark
Email: mengel@id.dtu.dk, rischel@id.dtu.dk

The talk proposes a solution to the Lamport-Broy specification problem for the Dagstuhl-Seminar 9439 using the Duration Calculus which is real-time interval temporal logic with a duration concept, and using the specification language **Z** as mathematical notation.

The Duration Calculus was originally invented as a notation for specifying requirements and high-level design for hybrid systems. It is a real-time interval temporal logic based on a dynamical systems model with states as functions of (real-valued) time. The notation provides a convenient way of expressing properties about durations of states and their temporal sequencing, and it has been applied to a number of examples. It is, hovewer, not possible to express an abstract liveness property (i.e. that something will eventually happen), so the notation is not appropriate for the Dagstuhl problem — unless abstract liveness is replaced by explicit time bounds.

The talk reports on an ongoing effort to develop a generalisation of the Duration Calculus, called Signed Duration Calculus (SDC) which allows us to

[1] On leave from Institute of Informatics, Warsaw University, Poland

specify abstract liveness property. The SDC notation was found only very recently and has not yet been throughly investigated. The specification for the Dagstuhl problem is in fact a part of our experiments with the new notation.

Modal Transition Systems and Abstraction

Kim Larsen, Aalborg University Center,
Bernhard Steffen, University of Passau,
Carsten Weise, Aachen University of Technology
Email: steffen@fmi.uni-passau.de

Our memory specification is based on the framework of (Timed) Modal Transitions Systems (MTS), which are labelled transition systems with two kinds of transitions: *may*- and *must*-transitions. Intuitively, must-transitions pose *requirements*, whereas may-transition specify the maximum transition *potential* of a possible implementation. An MTS is consistent if all required transitions are allowed. Satisfaction of a specification is formalized as a (weak) refinement relation. Intuitively, a refinement of a specification preserves all requirements, but it may eliminate of require allowed transitions. This notion of refinement can be extended to capture the nature of internal behaviour in the usual fashion, and timed behaviour can be modelled using real numbers as labels.

Our approach is characterized by its fine granularity: each ground instance of an activity, which is characterized by the kind of action, the calling processor, the considered location and, if necessary, the value of interest, is specified by means of its own finite state process. These very simple 'projective views' are then composed via (infinite) conjunction and parallel composition to the overall specification. This specification format allows modular correctness proofs on the basis of general algebraic laws, abstract interpretation and Skolemization. In fact, after algebraic simplification and exploitation of the limitations of the value dependencies of the memory specification for abstraction, the whole verification of the untimed case boils down to eight verification steps considering processes with less than ten states. These have been automatically performed on the TAV system. Of course, the timed case additionally needs an abstraction of the, in our case dense, time domain. Using the technique of timer region graphs, the timed case could be autimatically verified on the EPSILON system.

The Dagstuhl Problem — a Relational Approach

Ketil Stølen
Institut für Informatik, Technische Universität München
Email: stoelen@informatik.tu-muenchen.de

The memory component is specified in a relational style, namely by characterizing the allowed relation between the communication histories of the input and the output channels. The communication history of a channel is modeled by a stream. A stream is a (possibly infinite) sequence of actions, where each action models a message sent along the channel.

The memory component is specified in four steps. First a sequential memory is specified. This specification is then generalized to allow:

- concurrent calls,
- calls that result in several memory accesses,
- calls that fail.

The specification of the memory component is decomposed into a network of relational specifications characterizing an implementation based on remote procedure call. The required proof-obligation is stated and its verification is discussed.

A Fully Automated Logical Analysis of a Problem by Broy & Lamport

Kim Sunesen, Nils Klarlund, and Mogens Nielsen
Basic **R**esearch **i**n **C**omputer **S**cience,
Centre of the Danish National Research Foundation
Department of Computer Science, University of Aarhus
Email: ksunesen@daimi.aau.dk

We use monadic second-order logic (M2L) to solve a specification problem by Broy and Lamport. M2L is a succinct expression of regularity that so far has been only litte explored for specification and verification.

We propose M2L as an attractive logic for defining specifications, refinement mappings, and temporal properties within one framework. In fact, many requirements from the Broy and Lamport document can be translated sentence by sentence into M2L.

We develop techniques for expressing refinement mappings as temporal predicates. Thus we avoid the introduction of the usual history or prophecy variables.

Another contribution of this work is a new technique for reducing fairness properties to logical properties of finite sequences. This enables us to verify that fairness is preserved by verifying a property in M2L.

Finally, we show that our techniques can in fact be carried out in practice by an existing decision procedure for M2L. All aspects of the Broy and Lamport article have been explored (except real-time) and analysed. Using the

counter-model facility of the M2L tool, we have found and corrected several inconsistencies while developing our specification.

In conclusion, our automated framework shows that for certain problems, the M2L approach achieves the conciseness of predicate logic but without the cost of user intervention in terms of lemmas and tactics.

The Dagstuhl Specification Problem UNITY - Refinement Calculus

Rob T. Udink[2] and Joost N. Kok
Department of Computer Science, Utrecht University
Email: {rob,joost}@cs.ruu.nl

This presentation deals with the "Problem Specification" of the seminar. The problem concerns the specification and refinement of a Memory component that communicates with its environment by a procedure-calling interface. In this presentation, the problem is solved using an extension of the UNITY framework of Chandy and Misra. The UNITY framework consists of a programming language, for modeling fair transition systems, and a temporal logic. We combine this framework with methods known for the Refinement Calculus: we extend UNITY with local variables and use (data) refinement principles of Back and von Wright, and we add procedures in a similar way as is done by Back and Sere. In this way we obtain a small set of simple, compositional program transformation rules that preserve temporal properties in arbitrary context. These rules are sufficient to prove the refinements of the specification problem.

[2]This research has been supported by the Foundation for Computer Science in the Netherlands SION under project 612-317-107.