A Case Study in Automated Verification Based on Trace Abstractions

Nils Klarlund
Mogens Nielsen
Kim Sunesen

BRICS Report Series RS-95-54

ISSN 0909-0878 November 1995

A Case Study in Verification Based on Trace Abstractions*

Nils Klarlund† Mogens Nielsen Kim Sunesen
BRICS‡
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C.
{klarlund,mnielsen,ksunesen}@daimi.aau.dk

Abstract

In [14], we proposed a framework for the automatic verification of reactive systems. Our main tool is a decision procedure, Mona, for Monadic Second-order Logic (M2L) on finite strings. Mona translates a formula in M2L into a finite-state automaton. We show in [14] how *traces*, i.e. finite executions, and their abstractions can be described behaviorally. These state-less descriptions can be formulated in terms of customized temporal logic operators or idioms.

In the present paper, we give a self-contained, introductory account of our method applied to the RPC-memory specification problem of the 1994 Dagstuhl Seminar on Specification and Refinement of Reactive Systems. The purely behavioral descriptions that we formulate from the informal specifications are formulas that may span 10 pages or more.

Such descriptions are a couple of magnitudes larger than usual temporal logic formulas found in the literature on verification. To securely write these formulas, we introduce Fido [16] as a reactive system description language. Fido is designed as a high-level symbolic language for expressing regular properties about recursive data structures.

All of our descriptions have been verified automatically by Mona from M2L formulas generated by Fido.

Our work shows that complex behaviors of reactive systems can be formulated and reasoned about without explicit state-based programming. With Fido, we can state temporal properties succinctly while enjoying automated analysis and verification.

*Previously announced under the title: Using Monadic Second-Order Logic with Finite Domains for Specification and Verification.

†Author's current address: AT&T Research, Room 2C-410, 600 Mountain Ave., Murray Hill, NJ 07974; E-mail: `klarlund@research.att.com`

‡**B**asic **R**esearch **in** **C**omputer **S**cience,
Centre of the Danish National Research Foundation.

Contents

1 Introduction

In *reactive systems*, computations are regarded as sequences of events or states. Thus programming and specification of such systems focus on capturing the sequences that are allowed to occur. There are essentially two different ways of defining such sets of sequences.

In the *state approach*, the state space is defined by declarations of program variables, and the state changes are defined by the program code.

In the *behavioral approach*, the allowed sequences are those that satisfy a set of temporal constraints. Each constraint imposes restrictions on the order or on the values of events.

The state approach is used almost exclusively in practice. State based descriptions can be effectively compiled into machine code. The state concept is intuitive, and it is the universally accepted programming paradigm in industry.

The behavioral approach offers formal means of expressing temporal or behavioral patterns that are part of our understanding of a reactive system. As such, descriptions in this approach are orthogonal to the state approach—although the two essentially can express the same class of phenomena.

In this paper, we pursue the purely behavioral approach to solve the RPC-memory specification problem [3] posed by Manfred Broy and Leslie Lamport in connection with the Dagstuhl Seminar on Specification and Refinement of Reactive Systems. The main part of the problem is to verify that a distributed system P *implements* a distributed system S, that is, that every behavior of P is a behavior of S. Both systems comprise a number of processes whose behaviors are described by numerous informally stated temporal requirements like "Each successful $\mathsf{Read}(l)$ operation performs a single atomic read to location l at some time between the call and return."

The behavioral approach that we follow is the one we formulated in [14]. This approach is based on expressing behaviors and their abstractions in a decidable logic. In the present paper, we give an introductory and self-contained account of the method as applied to the Dagstuhl problem.

We hope to achieve two goals with this paper:

- to show that the behavioral approach can be used for verifying complicated systems—whose descriptions span many pages of dense, but readable, logic—using decision procedures that require little human intervention; and
- to introduce the Fido language as an attractive means of expressing finite-state behavior of reactive systems. (Fido is a programming language designed to express regular properties about recursive data structures [16].)

An overview of our approach

Our approach is based on the framework for automatic verification of distributed systems that we described in [14]. There, we show how *traces*, ie. finite computations, can be characterized behaviorally. We use *Monadic Second-order Logic*

(M2L) on finite strings as the formal means of expressing constraints. This decidable logic expresses regular sets of finite strings, that is, sets accepted by finite-state machines. Thus, when the number of processes and other parameters of the verification problem is fixed, the set L_P, of traces of P can be expressed by finite-state machines synthesized from M2L descriptions of temporal constraints. Similarly, a description of the set L_S of traces of the specification can be synthesized.

The *verifier*, who is trying to establish that P implements S, cannot just directly compare L_P and L_S. In fact, these sets are usually incomparable, since they involve events of different systems. As is the custom, we call the events of interest the *observable events*. These events are common to both systems. The *observable behaviors* $\mathit{Obs}(L_P)$ of L_P are the traces of L_P with all non-observable events projected away. That P implements S means that $\mathit{Obs}(L_P) \subseteq \mathit{Obs}(L_S)$.

One goal of the automata-theoretic approach to verification is to establish $\mathit{Obs}(L_P) \subseteq \mathit{Obs}(L_S)$ by computing the product of the automata describing $\mathit{Obs}(L_P)$ and $\mathit{Obs}(L_S)$. Specifically, we let A_P be an automaton accepting $\mathit{Obs}(L_P)$ and we let A_S be a automaton representing the complement of $\mathit{Obs}(L_S)$. Then $\mathit{Obs}(L_P) \subseteq \mathit{Obs}(L_S)$ holds if and only if the product of A_P and A_S is empty. Unfortunately, the projection of traces may entail a significant blow-up in the size of A_S as a function of the size of the automaton representing L_S. The reason is that the automaton A_S usually can be calculated only through a subset construction.

The use of state abstraction mappings or homomorphisms may reduce such state space blow-ups. But the disadvantage to state mappings is that they tend to be specified at a very detailed level: each global state of P is mapped to a global state of S.

In [14], we formulate well-known verification concepts, like *abstractions* and *decomposition principles* for processes in the M2L framework. The resulting trace based approach offers some advantages to conventional state based methods.

For example, we show how trace abstractions, which relate a trace of P to a corresponding trace of S, can be formulated loosely in a way that reflects only the intuition that the verifier has about the relation between P and S—and that does not require a detailed, technical understanding of how every state of P relates to a state of S. A main point of [14] is that even such loose trace abstractions may (in theory at least) reduce the non-determinism arising in the calculation of A_S.

The framework of [14] is tied closely to M2L: traces, trace abstractions, the property of implementation, and decomposition principles for processes are all expressible in this logic—and thus all amenable, in theory at least, to automatic analysis, since M2L is decidable.

In the present paper, we have chosen the `Fido` language both to express our concrete model of the Dagstuhl problem and to formulate our exposition of the framework of [14]. `Fido` is a notational extension of M2L that incorporates traditional concepts from programming languages, like recursive data types, functions, and strongly typed expressions. `Fido` is compiled into M2L.

An overview of the Dagstuhl problem

The Specification Problem of the Dagstuhl Seminar on Specification and Refinement of Reactive Systems is a four page document describing interacting components in distributed memory systems. Communication between components takes place by means of procedures, which are modeled by call and return events. at the highest level, the specification describes a system consisting of a memory component that provides read and write services to a number of processes. These services are implemented by the memory component in terms of basic i/o procedures. The relationships among service events, basic events, and failures are described in behavioral terms.

Problem 1 in the Dagstuhl document calls for the comparison of this memory system with a version, where a certain type of memory failure cannot occur.

Problem 2 calls for a formal specification of another layer added to the memory system in form of an RPC (Remote Procedure Call) component that services read and write requests.

Problem 3 asks for a formal specification of the system as implemented using the RPC layer and a proof that it implements the memory system of Problem 1.

In addressing the problems, we deal with safety properties of finite systems.

Problems 4 and 5 address certain kinds of failures that are described in a real-time framework. Our model is discrete, and we have not attempted to solve this part.

Previous work

Tne TLA formalism by Lamport [19] and the temporal logic of Manna and Pnueli [23, 13] provide uniform frameworks for specifying systems and state mappings, and for complex reasoning about systems. Both logics subsumes predicate logic logic and hence defy automatic verification in general. However, work has been done on providing mechanical support in terms of proof checkers and theorem provers, see [8, 9, 22].

The use of state mappings have been widely advocated, see e.g. [20, 18, 19, 13] and for a survey [21]. The involved theory of state mappings applicable to possibly infinite-state systems was established in [1, 15, 25].

The Concurrency Workbench [6] offers automatic verification of the existence of certain kinds of state-mappings between finite-state systems.

Decomposition is a key aspect of any verification methodology. In particular, almost all the solutions of the RPC-memory specification problem [3] in [4] use some sort of decomposition. In [2], Lamport and Abadi gave a proof rule for compositional reasoning in an assumption/guarantee framework. A non-trivial decomposition of a closed system is achieved by splitting it into a number of open systems with assumptions reflecting their dependencies. In our rule, dependencies are reflected in the choice of trace abstractions between components and a requirement on the relationship between the trace abstractions.

For finite-state systems, the COSPAN [10] tool based on the automata-theoretic framework of Kurshan [17] implements a procedure for deciding language containment for ω-automatas.

In [5], Clarke, Browne, and Kurshan shows how to reduce the language containment problem for ω-automata to a model checking problem in the restricted case where the specification is deterministic. The SMV tool [24] implements a model checker for the temporal logic CTL [7]. In COSPAN and SMV, systems are specified using typed C-like programming languages.

In the rest of the paper

In Section 2, we first explain M2L and then introduce the `Fido` notation by an example. Section 3 and 4 discuss our framework and show how all concepts can be expressed in `Fido`. We present our solution to the RPC-memory specification problem [3] dealing with the safety properties of the untimed part in Sections 5 to 8.

Acknowledgements

We would like to thank the referees for their comments and remarks.

2 Monadic second-order logic on strings

The logical notations we use are based on the monadic second-order logic on strings (M2L). A closed M2L formula is interpreted relative to a natural number n (the *length*). Let $[n]$ denote the set $\{0, \ldots, n-1\}$. First-order variables range over the set $[n]$ (the set of *positions*), and second-order variables range over subsets of $[n]$. We fix countably infinite sets of first and second-order variables $\mathit{Var}_1 = \{p, q, p_1, p_2, \ldots\}$ and $\mathit{Var}_2 = \{P, P_1, P_2, \ldots\}$, respectively. The *syntax* of M2L formulas is defined by the abstract syntax:

$$\begin{array}{lcl} t & ::= & p < q \mid p \in P \\ \phi & ::= & t \mid \neg\phi \mid \phi \vee \phi \mid \exists p.\phi \mid \exists P.\phi \end{array}$$

where p,q and P range over Var_1 and Var_2, respectively.

The standard *semantics* is defined as follows. An M2L formula ϕ with free variables is interpreted relative to a natural number n and an interpretation (partial function) $\mathcal{I}$ mapping first and second-order variables into elements and subsets of $[n]$, respectively, such that $\mathcal{I}$ is defined on the free variables of ϕ. As usual, $\mathcal{I}[a \leftarrow b]$ denotes the partial function that on c yields b if $a = c$, and otherwise $\mathcal{I}(c)$. We define inductively the *satisfaction relation* $\models_\mathcal{I}$ as follows.

$$\begin{array}{lllcc}
n & \models_\mathcal{I} & p < q & \stackrel{\text{def}}{\iff} & \mathcal{I}(p) < \mathcal{I}(q) \\
n & \models_\mathcal{I} & p \in P & \stackrel{\text{def}}{\iff} & \mathcal{I}(p) \in \mathcal{I}(P) \\
\\
n & \models_\mathcal{I} & \neg\phi & \stackrel{\text{def}}{\iff} & n \not\models_\mathcal{I} \phi \\
n & \models_\mathcal{I} & \phi \vee \psi & \stackrel{\text{def}}{\iff} & n \models_\mathcal{I} \phi \vee n \models_\mathcal{I} \psi \\
\\
n & \models_\mathcal{I} & \exists p.\phi & \stackrel{\text{def}}{\iff} & \exists k \in [n].n \models_{\mathcal{I}[p\leftarrow k]} \phi \\
n & \models_\mathcal{I} & \exists P.\phi & \stackrel{\text{def}}{\iff} & \exists K \subseteq [n].n \models_{\mathcal{I}[P\leftarrow K]} \phi
\end{array}$$

As defined above M2L is rich enough to express the familiar atomic formulas such as successor $p = q + 1$, as well as formulas constructed using the Boolean connectives such as $\wedge, \Rightarrow$ and $\Leftrightarrow$, and the universal first and second-order quantifier $\forall$, following standard logical interpretations. Throughout this paper we freely use such M2L derived operators.

There is a standard way of associating a language over a finite alphabet with an M2L formula. Let $\alpha = \alpha_0 \dots \alpha_{n-1}$ be a string over the alphabet $\{0,1\}^l$. Then the length $|\alpha|$ of α is n and $(\alpha_j)_i$ denotes the ith component of the l-tuple denoted by α_j. An M2L formula ϕ with free variables among the second-order variables $P_1, \dots, P_l$ defines the language:

$$L(\phi) = \{\alpha \in (\{0,1\}^l)^* \mid |\alpha| \models_{\mathcal{I}_\alpha} \phi\}$$

of strings over the alphabet $\{0,1\}^l$, where $\mathcal{I}_\alpha$ maps P_i to the set $\{j \in [n] \mid (\alpha_j)_i = 1\}$.

Any language defined in this way by an M2L formula is regular; conversely, any regular language over $\{0,1\}^l$ can be defined by an M2L formula. Moreover, given an M2L formula ϕ a minimal finite automaton accepting $L(\phi)$ can effectively be constructed using the standard operations of product, subset construction, projection, and minimization. This leads to a decision procedure for M2L, since ϕ is a tautology if and only if $L(\phi)$ is the set of all strings over $\{0,1\}^l$. The approach extends to any finite alphabet. For example, letters of the alphabet $\Sigma = \{a,b,c,d\}$ are encoded by letters of the alphabet $\{0,1\}^2$ by enumeration: a, b, c and d are encoded by $(0,0)$, $(1,0)$, $(0,1)$ and $(1,1)$, respectively. Thus, any language over Σ can be represented as a language over $\{0,1\}^2$ and hence any regular language over Σ is the language defined by some M2L formula with two free second-order variables P_1 and P_2. For example, the formula ϕ:

$$\forall p.p \notin P_1 \wedge p \notin P_2$$

defines the language $\{a\}^*$, that is, $L(\phi) = \{(0,0)\}^*$. In particular since $L(\phi)$ is not the set of all strings over $\{0,1\}^2$, ϕ is not a tautology and any string not in $L(\phi)$ yields a length and an interpretation falsifying ϕ.

2.1 Fido

As suggested above, any regular language over any finite alphabet can be defined as the language of an open M2L formula by a proper encoding of letters as bit patterns, that is, by enumerating the alphabet. In our initial solution to the Dagstuhl problem, we did the encoding "by hand" using only the Unix m4 macro processor to translate our specifications into M2L; this is an approach we cannot recommend, since even minor syntactic errors are difficult to find. The Fido notation helps us overcome these problems. Below, we explain the Fido notation by examples introducing all needed concepts one by one.

Consider traces, i.e. finite strings, over an alphabet Event consisting of events Read and Return with parameters that take on values in finite domains and the event τ. A Read may carry one parameter over the domain $\{l_0, l_1, l_2\}$, and a Return may carry two parameters, one from the domain $\{v_0, v_1\}$, and one from the domain $\{\mathsf{normal}, \mathsf{exception}\}$. In Fido, the code:

```
type Loc      = l0,l1,l2;
type Value    = v0,v1;
type Flag     = normal,exception;
```

declares the enumeration types Value, Flag, and Loc. They define the domains of constants $\{l_0, l_1, l_2\}$, $\{v_0, v_1\}$, and $\{\mathsf{normal}, \mathsf{exception}\}$, respectively. The type definitions:

```
type Read   = Loc;
type Return = Value & Flag;
```

declare a new name Read for the type Loc and the record type Return, which defines the domain of tuples $\{[v, f] \mid v \in \mathsf{Value} \wedge f \in \mathsf{Flag}\}$. The alphabet Event is the union of Read, Return and $\{\tau\}$:

```
type Event = Read | Return | τ;
```

The union is a disjoint union by default, since the Fido type system requires the arguments to define disjoint domains. The types presented so far all define finite domains. Fido also allows the definition of recursive data types. For our purposes recursively defined types are of the form:

```
type Trace = Event(next: Trace) | empty;
```

Thus, Trace declares the infinite set of values $\{e_1 e_2 \ldots e_n \mathsf{empty} \mid e_i \in \mathsf{Event}\}$. In other words, the type Trace is the set of all finite strings of parameterized events in Event with an empty value added to the end. The details of coding the alphabet of events in second-order M2L variables is left to the Fido compiler.

Fido provides (among others) four kinds of variables ranging over *strings*, *positions*, *subsets of positions* and *finite domains*, respectively. The Fido code:

```
string γ: Trace;
```

declares a free variable γ holding an element (a string) of Trace. We often refer to γ just as a string.

A first-order variable p may be declared to range over all positions in the string γ by the Fido declaration:

pos p: γ;

Similarly, a second-order variable P ranging over subsets of positions of the string can be declared as:

set P: γ;

A variable event holding an element of the finite domain Event is declared by:

dom event: Event;

The Fido notation includes, besides M2L syntax for formulas, existential and universal quantification over all the kinds of variables and more. We introduce additional syntax when used. For example, we can specify as a formula that the event Read:[l_0] from the domain Event occurs in γ:

∃**pos** p :γ.(γ(p) = Read:[l_0])

which is true if and only if there exists a position p in γ such that the pth element in γ is the event Read:[l_0].

If we want to refer to a Read event without regard to the value of its parameter, we write:

∃**pos** p :γ;**dom** l: Loc.(γ(p) = Read:[l?])

which is true if and only if there exists a position p in γ and an element l in Loc such that the pth element in γ is the event Read:[l]. To make the above formula more succinct, we can use the pattern matching syntax of Fido, where a "dont't care" value is specified by a question mark:

∃**pos** p :γ.(γ(p) = Read:[?])

The Fido compiler translates such question marks into explicit existential quantifications over the proper finite domain.

A Fido *macro* is a named formula with type-annotated free variables. Below, we formulate some useful temporal concepts in Fido that formalize high-level properties of intervals. In the rest of the paper, we use strings to describe behaviors over time and therefore we refer to positions in strings as time instants in traces.

To say that a particular event event of type Event occurred before a given time instant t in trace α of type Trace, we write:

```
func Before(string α: Trace; pos t: α; dom event: Event): formula;
  ∃pos time: α.(time<t ∧ α(time)=event)
end;
```

To express that event occur sometime in the interval from t_1 to t_2 (both excluded), we write:

```
func Between(string α: Trace; pos t1,t2: α; dom event: Event): formula;
  ∃pos time: α.(t1 <time ∧ time<t2 ∧ α(time)=event)
end;
```

The property that in a trace γ a Return is always preceded by a Read is expressed as:

```
∀pos t: γ.(γ(t)=Return:[?,?] ⇒ Before(γ,t,Read:[?]));
```

We can also express that a Return event occurs exactly once in an interval:

```
func ExactlyOneReturnBetween(string α: Trace; pos t1,t2: α): formula;
  ∃pos time: α.(t1<time ∧ time<t2 ∧ α(time)=Return:[?,?] ∧
                ¬Between(α,t1,time,Return:[?,?]) ∧
                ¬Between(α,time,t2,Return:[?,?])
end;
```

That a Read event occurred at both end points of the interval, but not in the interval, is expressed as:

```
func ConseqReads(string α: Trace; pos t1,t2: α): formula;
  t1<t2 ∧ α(t1) =Read:[?] ∧ α(t2)=Read:[?] ∧
  ¬Between(α,t1,t2,Read:[?])
end;
```

Using the macros above it is easy to specify more complicated properties. For example, to specify that a Read event is blocking, in the sense that any Return is issued in response to a unique Read event and no two read events occurs consecutively without a return in between, we write:

```
func ReadProcs(string α: Trace): formula;
  ∀pos t1: α.
    α(t1)=Return:[?,?]
   ⇒
    ∃pos t0: α.(t0 <t1 ∧ α(t0)=Read:[?] ∧
      ¬Between(α,t0,t1, Return:[?,?])) ∧
  ∀pos time1,time2: α.
    ConseqReads(α,time1,time2)
   ⇒
    ExactlyOneReturnBetween(α,time1,time2)
end;
```

Finally in our Fido overview, we mention that strings may be quantified over as well. For example, the formula:

```
∃string α: γ;pos t: γ. (γ(t)=α(t));
```

expresses that there is a string α of the same type and length as γ and some time instant t in γ (and therefore also in α) such that the tth element of γ and α, respectively, are the same.

2.2 Automated translation and validity checking

Any well-typed Fido formula is translated by the Fido compiler [16] into an M2L formula. Hence, the Fido compiler together with the Mona tool [11] provides automatic verification, in terms of deciding whether or not a given

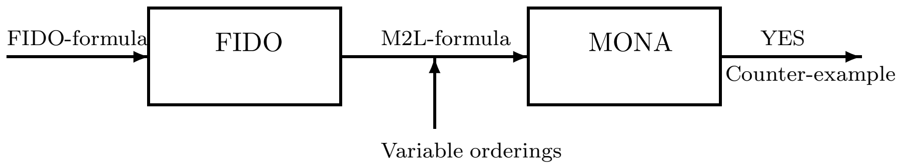

Figure 1: The Fido and Mona tools.

Fido input translates into a valid M2L formula, see Fig. 1. Furthermore, in the negative case, a witness in terms of a minimal interpretation falsifying the translation of ϕ is provided, and translated back to Fido level from the (minimal deterministic) automaton recognizing $L(\phi)$.

We will not describe the efficient translation of the high-level syntax of Fido into M2L formulas here. Instead, we emphasize that the translation is in principle straightforward: a string over a finite domain D is encoded using as many second-order variables (bits) as necessary to enumerate $\mathsf{D} \cup \{\mathsf{empty}\}$, quantification over strings amounts to quantification over the second-order variables encoding the alphabet, and existential (universal) quantification over finite domains amounts to a finite disjunction (conjunction) over the elements of the domain.

The Mona tool provides an efficient implementation of the underlying M2L decision procedure [11]. Since the implementation is based on BDD representations of automatas, it, importantly, allows formulas to be decorated with variable orderings.

3 Systems

We reason about computing systems through specifications of their behaviors in Fido, i.e. viewed as traces over parameterized events specified in terms of Fido formulas.

A system A determines an alphabet Σ_A of *events*, which is partitioned into *observable events* Σ_A^{Obs} and *internal events* Σ_A^{Int}. It is the observable events that matters when systems are compared. A *behavior* of A is a finite sequence over Σ_A. The system A also determines a prefix–closed language L_A of behaviors called *traces* of A. We write $A = (L_A, \Sigma_A^{Obs}, \Sigma_A^{Int})$. The *projection* π from a set Σ^* to a set Σ'^* ($\Sigma' \subseteq \Sigma$) is the unique string homomorphism from Σ^* to Σ'^* given by $\pi(a) = a$, if a is in Σ' and $\pi(a) = \epsilon$ otherwise, where ϵ is the empty string. The *observable behaviors* of a system A, $Obs(A)$, are the projections onto Σ_A^{Obs} of the traces of A, that is $Obs(A) = \{\pi(\alpha) \mid \alpha \in L_A\}$, where π is the projection from Σ_A^* onto $(\Sigma_A^{Obs})^*$.

A system A is thought of as existing in a *universe* of the systems with which it may be composed and compared. Formally, the universe is a global alphabet $\mathcal{U}$, which contains Σ_A and all other alphabets of interest. Moreover, $\mathcal{U}$ is assumed to contain the distinguished event τ which is not in the alphabet of any system. The set $N_\Sigma(A)$ of *normalized traces* over an alphabet $\Sigma \supseteq \Sigma_A$

is the set $h^{-1}(L_A) = \{\alpha \mid h(\alpha) \in L_A\}$, where h is the projection from Σ^* onto Σ_A^*. Normalization plays an essential rôle when composing systems and when proving correctness of implementation of systems with internal events.

A systems can conveniently be expressed in Fido. Following the discussion in Section 2 a finite domain U representing the universal alphabet $\mathcal{U}$, and a data type, $\mathsf{Trace}_\mathsf{U}$, representing the traces over U are defined. A system $A = (L_A, \Sigma_A^{Obs}, \Sigma_A^{Int})$ is then represented by a triple:

$$\mathsf{A} = (\mathsf{Norm}_\mathsf{A},\mathsf{Obs}_\mathsf{A},\mathsf{Int}_\mathsf{A})$$

of macros defining the normalized traces, Norm_A, of A over U, the observable events, Obs_A, and the internal events, Int_A. That is, let γ be a string over $\mathsf{Trace}_\mathsf{U}$ then $\mathsf{Norm}_\mathsf{A}(\gamma)$ is true if and only if γ denotes a trace of $N_{\mathcal{U}}(A)$ and let u be an element of U then $\mathsf{Obs}_\mathsf{A}(\mathsf{u})$ and $\mathsf{Int}_\mathsf{A}(\mathsf{u})$ are true if and only if u denotes an element of Σ_A^{Obs} and Σ_A^{Int}, respectively. When writing specifications in Fido, we often confuse the name of a system with the name of the macro defining its set of normalized traces.

Our first example of a system in Fido is the system ReadProcs living in the universe given by Event from Section 2. The normalized traces of ReadProcs are defined by the macro ReadProcs, the alphabet of observable events by:

```
func ObsReadProcs(dom v: Event; dom id: Ident): formula;
  v=Read:[?] ∨ v=Return:[?,?]
end;
```

and the alphabet of internal events by:

```
func IntReadProcs(dom v: Event; dom id: Ident): formula;
  false
end;
```

That is, ReadProcs has observable events Read:[?] and Return:[?,?], and no internal events:

$$\mathsf{ReadProcs} = (\mathsf{ReadProcs},\mathsf{ObsReadProcs},\mathsf{IntReadProcs})$$

3.1 Composition

Our notion of composition of systems is that of CSP [12], adjusted to cope with observable and internal events. We say that systems A and B are *composable* if they agree on the partition of events, that is, if no internal event of A is an observable event of B and vice versa, or symbolically, if $\Sigma_A^{Int} \cap \Sigma_B^{Obs} = \emptyset$ and $\Sigma_B^{Int} \cap \Sigma_A^{Obs} = \emptyset$. Given composable systems A and B, we define their *composition* $A \parallel B = (L_{A\parallel B}, \Sigma_{A\parallel B}^{Obs}, \Sigma_{A\parallel B}^{Int})$, where

- the set of observable events is the union of the sets of observable events of the components, that is, $\Sigma_{A\parallel B}^{Obs} = \Sigma_A^{Obs} \cup \Sigma_B^{Obs}$,
- the set of internal events is the union of the sets of internal events of the components, that is, $\Sigma_{A\parallel B}^{Int} = \Sigma_A^{Int} \cup \Sigma_B^{Int}$, and

- the set of traces is the intersection of the sets of normalized traces with respect to the alphabet $\Sigma_{A\|B}$, that is, $L_{A\|B} = N_{\Sigma_{A\|B}}(A) \cap N_{\Sigma_{A\|B}}(B)$.

As in CSP, a trace of $A \parallel B$ is the interleaving of a trace of A with a trace of B in which common events are synchronized. Composition is commutative, idempotent and associative, and we adopt the standard notation, $A_1 \parallel \ldots \parallel A_n$ or just $\parallel A_i$, for the composition of n composable systems A_i.

In Fido, composability of A and B is expressed by:

```
∀pos t:γ. (Int_A(γ(t)) ⇒ ¬ Obs_B(γ(t))) ∧ (Int_B(γ(t)) ⇒ ¬ Obs_A(γ(t)))
```

and given composable systems A and B, composition is defined by:

$$\mathsf{A} \parallel \mathsf{B} = (\mathsf{Norm}_{\mathsf{A}\|\mathsf{B}}, \mathsf{Obs}_{\mathsf{A}\|\mathsf{B}}, \mathsf{Int}_{\mathsf{A}\|\mathsf{B}})$$

where the set of normalized traces are defined by conjunction:

```
func Norm_A||B(string α: Trace_U): formula;
  Norm_A(α) ∧ Norm_B(α)
end;
```

and the alphabets by disjunction:

```
func Obs_A||B(dom v: U): formula;
  Obs_A(v) ∨ Obs_B(v)
end;

func Int_A||B(dom v: U): formula;
  Int_A(v) ∨ Int_B(v)
end;
```

To exemplify composition, we extend the universe Event with the events given by:

```
type Mem = Loc & Value & Flag;
```

Hence, the type Event is now:

```
type Event = Mem | Read | Return | τ;
```

The macro:

```
func MemBetween(string α: Trace): formula;
  ∀ dom l: Loc;dom v: Value;pos t1,t2: α.
      α(t1)=Read:[l?] ∧ α(t2)=Return:[v?,?]
    ⇒
      ∃pos t0: α. t0<t1 ∧ α(t0)=Mem:[l?,v?,?]
end;
```

is true on a trace if and only if there exists an atomic read event Mem:[l,v,?] between any read event Read:[l] to location l and return event Return:[v,?] with value v. We define the system MemBetween with observable events Read:[?] and Return:[?,?], and internal events Mem:[?,?,?]:

$$\mathsf{MemBetween} = (\mathsf{MemBetween}, \mathsf{ObsMemBetween}, \mathsf{IntMemBetween})$$

where

```
func ObsMemBetween(dom v: Event; dom id: Ident): formula;
  v=Read:[?] ∨ v=Return:[?,?]
end;
```

and

```
func IntMemBetween(dom v: Event; dom id: Ident): formula;
  Mem:[?,?,?]
end;
```

The systems ReadProcs and MemBetween are composable since they do not disagree on the partition of their alphabets. We define their composition:

$$\mathsf{MReadProcs} = \mathsf{ReadProcs} \parallel \mathsf{MemBetween}$$

Hence, MReadProcs has observable events Read:[?] and Return:[?,?], and internal events Mem:[?,?,?], and the normalized traces of MReadProcs specify the behaviors of read procedure calls with atomic reads.

3.2 Implementation

We formalize the notion of implementation in terms of language inclusion, again adjusted to cope with observable and internal events. We say that systems A and B are *comparable* if they have the same set of observable events Σ^{Obs}, that is, $\Sigma^{Obs} = \Sigma^{Obs}_A = \Sigma^{Obs}_B$. In the following A and B denote comparable systems with $\Sigma^{Obs}_A = \Sigma^{Obs}_B = \Sigma^{Obs}$.

Definition 1 Let A and B denote comparable systems. *A implements B* if and only if $\mathit{Obs}(A) \subseteq \mathit{Obs}(B)$

In Fido, comparability between systems is easily expressible:

$$\forall \mathbf{pos}\ \mathsf{t}{:}\gamma.\mathsf{Obs_A}(\gamma(\mathsf{t})) \Leftrightarrow \mathsf{Obs_B}(\gamma(\mathsf{t})) \tag{1}$$

Implementation is less obvious. One sound approach is to attempt a proof of $N_{\mathcal{U}}(A) \subseteq N_{\mathcal{U}}(B)$, which is easily expressible in Fido as the formula $\mathsf{Norm}_A(\gamma) \Rightarrow \mathsf{Norm}_B(\gamma)$. However, when the systems A and B have different internal behaviors the approach does not work in general.

Consider our example systems from above, we define the system

$$\mathsf{RMReadProcs} = (\mathsf{RMReadProcs}, \mathsf{ObsRMReadProcs}, \mathsf{IntRMReadProcs})$$

specifying reliable read procedures, that is, read procedures that never triggers exceptional atomic reads, where ObsRMReadProcs and IntRMReadProcs are equivalent to ObsMReadProcs and IntMReadProcs, respectively, and

```
func RMReadProcs(string α: Trace): formula;
  MReadProcs(α) ∧ ¬∃pos t: α.α(t)=Mem:[?,?,exception]
end;
```

The systems RMReadProcs and ReadProcs are comparable as they have the same set of observable events and the first implements the second since the implication:

$$\mathsf{RMReadProcs}(\gamma) \Rightarrow \mathsf{ReadProcs}(\gamma)$$

holds for all traces γ over Trace. The opposite implication does not hold, a simple counterexample is the trace Read:[$\mathsf{l_0}$] Mem:[$\mathsf{l_0}$,$\mathsf{v_0}$,exception] Return:[$\mathsf{v_0}$,normal] empty. However, the observable behaviors of the systems RMReadProcs and ReadProcs are clearly identical. In the next section, we show how to prove the implementation property using Fido.

4 Relational trace abstractions

A *trace abstraction* is a relation on traces preserving observable behaviors. In the following A and B denote comparable systems with $\Sigma_A^{Obs} = \Sigma_B^{Obs} = \Sigma^{Obs}$ and π denotes the projection of $\mathcal{U}^*$ onto $(\Sigma^{Obs})^*$.

Definition 2 [14] A trace abstraction $\mathcal{R}$ from A to B is a relation on $\mathcal{U}^* \times \mathcal{U}^*$ such that:

1. If $\alpha\mathcal{R}\beta$ then $\pi(\alpha) = \pi(\beta)$
2. $N_{\mathcal{U}}(A) \subseteq \mathit{dom}\, \mathcal{R}$
3. $\mathit{rng}\, \mathcal{R} \subseteq N_{\mathcal{U}}(B)$

The first condition states that any pair of related traces must agree on observable events. The second and third condition require that any normalized trace of A should be related to some normalized trace of B, and only to normalized traces of B.

Theorem 3 [14] There exists a trace abstraction from A to B if and only if A implements B.

Hence, the search for a trace abstraction is a sound and complete technique for deciding implementation. In the following, we incorporate the technique in the Fido framework.

Given strings $\alpha = \alpha_0 \ldots \alpha_n \in \Sigma_1^*$ and $\beta = \beta_0 \ldots \beta_n \in \Sigma_2^*$, we write $\alpha\hat{\ }\beta$ for the string $(\alpha_0, \beta_0) \ldots (\alpha_n, \beta_n) \in (\Sigma_1 \times \Sigma_2)^*$. Every language $L_{\mathcal{R}}$ over a product alphabet $\Sigma_1 \times \Sigma_2$ has a canonical embedding as a relation $\mathcal{R}_L \subseteq \Sigma_1^* \times \Sigma_2^*$ on strings of equal length given by $\alpha\hat{\ }\beta \in L_{\mathcal{R}} \overset{\text{def}}{\Leftrightarrow} \alpha\mathcal{R}_L\beta$. We say that a trace abstraction is *regular* if it is the embedding of a regular language over $\mathcal{U} \times \mathcal{U}$.

Not all trace abstractions between finite-state systems are regular. However, to use Fido we have to restrict ourselves to regular abstractions.

Definition 4 Given a subset Σ' of Σ, we say that strings $\alpha, \beta \in \Sigma^*$ are Σ'–*synchronized* if they are of equal length and if whenever the ith position in α contains a letter in Σ' then the ith position in β contains the same letter, and vice versa.

The property of being Σ^{Obs}-synchronized is Fido expressible:

```
func Observe(string $\alpha$: Trace$_U$; string $\beta$: $\alpha$): formula;
  ∀pos t: $\alpha$.(Obs$_A$($\alpha$(t)) ∨ Obs$_B$($\beta$(t))  ⇒ $\alpha$(t)= $\beta$(t))
end;
```

Definition 5 Let $\hat{\mathcal{R}}$ be the language over $\mathcal{U} \times \mathcal{U}$ given by $\alpha^\wedge\beta \in \hat{\mathcal{R}}$ if and only if

$$\beta \in N_{\mathcal{U}}(B) \text{ and } \alpha, \beta \text{ are } \Sigma^{Obs}\text{-synchronized}$$

Since $N_{\mathcal{U}}(B)$ is a regular language, so is $\hat{\mathcal{R}}$, and furthermore it may be expressed in Fido by:

```
func R(string $\alpha$: Trace$_U$; string $\beta$: $\alpha$): formula;
  Observe($\alpha$, $\beta$)  ∧  Norm$_B$($\beta$)
end;
```

The next proposition gives a sufficient condition for $\hat{\mathcal{R}}$ and any regular subset of $\hat{\mathcal{R}}$ to be a trace abstraction. We return to the significance of the last part when dealing with automated proofs.

Proposition 6 [14] If $N_{\mathcal{U}}(A) \subseteq dom\ \hat{\mathcal{R}}$ then $\hat{\mathcal{R}}$ is a regular trace abstraction from A to B. Moreover in general, for any regular language $\mathcal{C} \subseteq (\mathcal{U} \times \mathcal{U})^*$, if $N_{\mathcal{U}}(A) \subseteq dom\ \hat{\mathcal{R}} \cap \mathcal{C}$, then $\hat{\mathcal{R}} \cap \mathcal{C}$ is a regular trace abstraction from A to B.

Importantly, also prerequisites of this proposition may be expressed in Fido, and hence validity checking:

```
Norm$_A$($\alpha$) ⇒ ∃string $\beta$: $\alpha$.R($\gamma$,$\beta$)
```

is a sound and fully automated (!) technique for deciding implementation.

To prove that the system MReadProcs implements RReadProcs we instantiate macro Observe and R properly, and then check that:

```
MReadProcs($\gamma$) ⇒ ∃string $\beta$: $\alpha$.R($\gamma$,$\beta$)
```

holds.

4.1 Decomposition

One thing is to have a sound proof technique, another is to have an efficient automated implementation of it. It is well known that compositional reasoning is one important way of obtaining efficiency, and one important aspect of trace abstractions is that they allow compositional reasoning, in the following formal sense [14].

Theorem 7 [14] Let A_i and B_i be pairwise comparable systems forming the compound systems $\| A_i$ and $\| B_i$. If

$$\mathcal{R}_i \text{ is a trace abstraction from } A_i \text{ to } B_i. \tag{2}$$

$$\bigcap_i \ dom\, \mathcal{R}_i \subseteq \ dom\, \bigcap_i \mathcal{R}_i \tag{3}$$

then

$$\|A_i \ implements \ \|B_i$$

We call the extra condition (3) the *compatibility requirement.* By allowing components of a compound systems to also interact on internal events, we allow systems to be non-trivially decomposed. This is why the compatibility requirement (3) is needed, intuitively, it ensures that the choices defined by the trace abstractions can be made to agree on shared internal events. Formally, the intuition is expressed by the corollary:

Corollary 4.1 [14] If additionally the components of the specification are non-interfering on internal events, that is, $\Sigma_{B_i}^{Int} \cap \Sigma_{B_j}^{Int} = \emptyset$, for every $i \neq j$, then A_i implements B_i implies $\|A_i$ implements $\|B_i$.

Again, the compatibility requirement is expressible in Fido:

$$\textstyle\bigwedge_{\mathsf{i}=1,\dots,\mathsf{n}} \ (\exists \textbf{string } \beta_{\mathsf{i}}\text{: } \gamma.(\mathsf{R}_{\mathsf{i}}(\gamma,\beta_{\mathsf{i}}))) \Rightarrow \exists \textbf{string } \beta\text{: } \gamma.(\bigwedge_{\mathsf{i}=1,\dots,\mathsf{n}} \mathsf{R}_{\mathsf{i}}(\gamma,\beta)) \tag{4}$$

where R_i is a Fido macro taking as parameters two strings of type Trace and n is some fixed natural number.

The use of Theorem 7 for compositional reasoning about non-trivial decompositions of systems is illustrated in Section 8.

5 The RPC-memory specification problem

The rest the paper describes our solution to the RPC-memory specification problem proposed by Broy and Lamport [3] considering the safety properties of the untimed part. In the hope of improved readability and comparability we choose to copy into the text parts of the informal description in small pieces printed in *italic.*

5.1 The procedure interface

The problem [3] calls for the specification and verification of a series of components interacting with each other using a procedure-calling interface. In our specification, components are systems defined by Fido formulas. Systems interact by synchronizing on common events - internal as well as observable - there is no notion of sender and receiver on this level. A procedure call consists of a *call* and the corresponding *return.* Both are indivisible (atomic) events. There are two kinds of returns, *normal* and *exceptional.* A component may contain a number of concurrent processes each carrying a unique identity. Any call or return triggered by a process communicates its identity. This leads us to declare the parameter domains:

```
type Flag  = normal,exception;
type Ident = id_0,...,id_k;
```

of return flags and process identities for some fixed k, respectively.

6 A memory component

The first part of the problem [3] calls for a specification of a memory component. The component should specify a memory that maintains the contents of a set MemLocs of locations such that the contents of a location is an element of a set MemVals. We therefore introduce the domains:

```
type MemLocs = l_0,...,l_n;
type MemVals  = initVal,v_1,...,v_m;
```

of locations and of values for some fixed n and m, respectively. The reason for defining the distinguished value initVal follows from: *The memory behaves as if it maintains an array of atomically read and written locations that initially all contain the value* InitVal*.* Furthermore, we accordingly define the following Mem events carrying five parameters.

```
type Mem = Operation & MemLocs & MemVals & Flag & Ident;
```

The first parameter defined by the domain:

```
type Operation = rd,wrt;
```

indicates whether the event denotes an atomic read or write operation. The second and third carry the location to be and the value read or written, respectively. The fourth indicates the success of the operation. We hence also allow atomic reads and writes to exhibit exceptional behavior. Finally, the fifth parameter carries a process identity (meant to indicate the identity of the process that triggered the event).

The component has two procedure calls: *reads* and *writes*. The informal description [3] notes 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.* As we aim for automatic verification the number of states as well as events are crucial. Hence, we try to be particular in not tacitly reducing any of these by faithfully modeling all possible erroneous events. Hence, we introduce the domains:

```
type Tag    = ok,error;
type Loc    = MemLocs & Tag;
type Value  = MemVals & Tag;
```

The idea is that procedure calls and returns pass arguments of type Loc and Value whose first components denote semantically correct elements of respectively MemLocs and MemVals if and only if the value of the corresponding Tag components are ok. In the informal description [3], a read procedure is described as:

Name	Read
Arguments	loc *: an element of* MemLocs
Return Value	*an element of* MemVals
Exception	BadArg *: argument loc is not an element of* MemLocs.
	MemFailure*: the memory cannot be read.*
Description	*Returns the value stored in address* loc.

In our specification, a read procedure is called by issuing a Read event of the type:

type Read = Loc & Ident & Visible;

A Read event takes as first parameter an element of Loc that might not be a "semantically" correct element of MemLocs. and as second parameter a process identity. The last parameter is an element of the domain:

type Visible = internal,observable;

When verifying the implementation we need the parameter Visible to be able to change the view of reads, writes and returns from observable to internal events.

The return of a read procedure in our specification is a Return event given by:

type Return = Value & Flag & RetErr & Ident & Visible;

The first parameter is the value returned. The second indicates whether the return is normal or exceptional. In case, it is exceptional the third parameter is an element of the domain:

type RetErr = BadArg,MemFailure;

of possible errors returned by an exceptional return as described above.

Again, the fourth and fifth parameter are elements of the domains Ident and Visible with the intended meaning as for Read events. Similarly, a write procedure is specified in terms of Write events defined by:

type Write = Loc & Value & Ident & Visible;

and Return events. Hence, the universe for our systems is given by:

type Event = Mem | Read | Write | Return | τ;

and traces (strings) over the universe by:

type Trace = Event(next: Trace) | empty;

We specify the memory component Spec by the compound system:

$$\mathsf{Spec} = \mathsf{MemSpec}(\mathsf{id}_0) \parallel \ldots \parallel \mathsf{MemSpec}(\mathsf{id}_k) \parallel \mathsf{InnerMem}$$

constructed from systems MemSpec(id) that specify read and write procedures for fixed process identities id and a system InnerMem that specifies the array maintained by the memory component. Each of the systems MemSpec(id) are themselves compound systems:

$$\mathsf{MemSpec}(\mathsf{id}) = \mathsf{ReadSpec}(\mathsf{id}) \parallel \mathsf{WriteSpec}(\mathsf{id})$$

defined by composing the systems ReadSpec(id) and WriteSpec(id) specifying respectively read and write procedures for fixed process identities id.

For a fixed process identity id in Ident, the system ReadSpec(id) with observable events Read:[?,id,observable] and Return:[?,?,?,id,observable] and internal events Mem:[rd,?,?,?,id] specifies the allowed behaviors of read procedure calls involving the process with identity id. In Fido notation, a *logical and* ($\wedge$) can alternatively be written as a semicolon (;). The normalized traces of ReadSpec(id) are defined by the macro:

```
func ReadSpec(string α: Trace; dom id: Ident; dom vis: Visible): formula;
  BlockingCalls(α,id,vis);
  CheckSuccessfulRead(α,id,vis);
  WellTypedRead(α,id,vis);
  ReadBadArg(α,id,vis);
  OnlyAtomReadsInReadCalls(α,id,vis)
end;
```

That is, γ is a normalized trace of ReadSpec(id) if and only if ReadSpec(γ,id,observable) is true. In the following, we often implicitly specialize macros, e.g. we write ReadSpec(id,observable) for the macro obtained from ReadSpec by instantiating the parameters id and vis. The system ReadSpec(id) is then given by the triple:

(ReadSpec(id,observable),ObsReadSpec(id,observable),IntReadSpec(id))

where

```
func ObsReadSpec(dom v: Event; dom id: Ident; dom vis: Visible): formula;
  v=Read:[?,id?,vis?] ∨ v=Return:[?,?,?,id?,vis?]
end;
```

and

```
func IntReadSpec(dom v: Event; dom id: Ident): formula;
  v=Mem:[rd,?,?,?,id?]
end;
```

The macro ReadSpec is the conjunction of five clauses. The first clause BlockingCalls specifies as required in [3] that procedure calls are blocking in the sense that a process stops after issuing a call and waits for the corresponding return to occur. The last clause OnlyAtomReadsInReadCalls specifies that an atomic read event occurs only during the handling of read calls. This requirement is not described in [3]. Reading in between the lines however, it seems clear that the specifier did not mean for atomic reads to happen without being part of some read procedure call. Both clauses are straightforwardly defined in Fido using interval temporal idioms similar to those explained in Section 2.1.

As we demonstrate below, the three mid clauses are defined as fairly direct transcriptions of the text of [3] describing read procedure calls. But first, a convenient macro definition. Following [3], an *operation* consists of a procedure call and the corresponding return. We define the macro:

```
func Opr(string $\alpha$: Trace; pos $t_1$,$t_2$: $\alpha$;
          dom call,return: Event; dom id: Ident; dom vis: Visible): formula;
  $t_1$<$t_2$ $\wedge$ $\alpha$($t_1$)=call $\wedge$ $\alpha$($t_2$)=return;
  $\neg$Between($\alpha$,$t_1$,$t_2$,Read:[?,id?,vis?]);
  $\neg$Between($\alpha$,$t_1$,$t_2$,Write:[?,?,id?,vis?]);
  $\neg$Between($\alpha$,$t_1$,$t_2$,Return:[?,?,?,id?,vis?])
end;
```

which is true for a trace γ, time instants t_1 and t_2 in γ and events call and return if and only if the events call and return occurred at t_1 and t_2, respectively, and none of the events Read, Write and Return occurred between t_1 and t_2 (both excluded). An operation is *successful* if and only if its return is normal (non-exceptional).

The lines (excluding the last one) quoted from [3] above describing a read procedure are translated into the following macro quantifying over both location and value tags, flags, return errors and time instants:

```
func WellTypedRead(string $\alpha$: Trace; dom id: Ident; dom vis: Visible): formula;
  $\forall$dom vt,lt: Tag; dom retErr: RetErr; dom flg: Flag; pos $t_1$,$t_2$: $\alpha$.
    Opr($\alpha$,$t_1$,$t_2$, Read:[[?,lt?],id?,vis?],Return:[[?,vt?],flg?,retErr?,id?,vis?],id,vis)
   $\Rightarrow$
    (flg=normal;lt=MemLocs;vt=MemVals) $\vee$
    (flg=exception;retErr=MemFailure) $\vee$
    (flg=exception;$\neg$lt=MemLocs;retErr=BadArg)
end;
```

establishing the connection among the parameters received and those returned. Whenever a read call and the corresponding return has occurred, then either the return was normal and the value as well as the location passed were of the right types (respectively MemVals and MemLocs) or the return was exceptional and the error returned was MemFailure or the return was exceptional and the location passed was not of the right type (MemLocs) and the returned error was BadArg.

Furthermore, it is stated in [3] that:

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

We transcribe this into the macro:

```
func ReadBadArg(string $\alpha$: Trace; dom id: Ident; dom vis: Visible): formula;
  $\forall$ pos $t_1$,$t_2$: $\alpha$.
    Opr($\alpha$,$t_1$,$t_2$,Read:[?,id?,vis?],Return:[?,exception,BadArg,id?,vis?],id,vis)
   $\Rightarrow$
    $\neg$Between($\alpha$,$t_1$,$t_2$,Mem:[?,?,?,?,id?])
end;
```

specifying that between the call and the return of a read operation resulting in an exceptional return with return error BadArg no atomic read or write is performed. (Note that we interpreted *no effect on the memory* as the absence of atomic reads and writes.)

Finally, a read procedure must satisfy that:

Each successful Read(l) *operation performs a single atomic read to location* l *at some time between the call and return.*

Together with the line excluded above we get that the value returned should be the value read in the atomic read. This relation between a successful read and the corresponding return is captured by the macro:

```
func CheckSuccessfulRead(string α: Trace; dom id: Ident; dom vis: Visible): formula;
  ∀ dom v: MemVals; dom l: MemLocs; dom flg: RetErr; pos t₁: α; pos t₂: α.
    (Opr(α,t₁,t₂, Read:[[l?,?],id?,vis?],Return:[[v?,ok],normal,?,id?,vis?],id,vis)
    ⇒
    ∃ pos time: α.
      (t₁ <time ∧ time<t₂ ∧ α(time)=Mem:[rd,l?,v?,normal,id?];
      ¬Between(α,t₁,time,Mem:[rd,?,?,?,id?]);
      ¬Between(α,time,t₂,Mem:[rd,?,?,?,id?])))
end;
```

requiring that if the return is normal (and thus the read successful) then exactly one atomic read is performed between the call and the return on the requested location. Furthermore, the value returned is the value read.

The systems WriteSpec(id) are defined similarly to the systems ReadSpec(id) though slightly more complicated since write calls carries more parameters. The observable events of WriteSpec(id) are Write:[?,id,observable] and Return:[?,?,?,id,observable], and the internal events are Mem:[wrt,?,?,?,id].

The system InnerMem defines the behaviors allowed by the array maintained by the memory component. The informal description [3] refers to but does not define an array. We apply the informal description: whenever a successful atomic read to a location occurs the value thus returned is the value last written by a successful atomic write on the location or if no such atomic write has occurred its the initial value initVal. The normalized traces of InnerMem are defined by the macro:

```
func InnerMem(string α: Trace): formula;
  ∀ dom v: MemVals; dom l: MemLocs; pos t: α.
     α(t)=Mem:[rd,l?,v?,normal,?]
    ⇒
     ∃ pos t₀: α.(t₀ <t ∧ α(t₀)=Mem:[wrt,l?,v?,normal,?] ∧
        ¬Between(α,t₀,t,Mem:[wrt,l?,?,normal,?])) ∨
    v=initVal ∧ ¬Before(α,t,Mem:[wrt,l?,?,normal,?])
end;
```

The system InnerMem has internal events Mem:[?,?,?,?,?] and no observable events and is hence given by the triple:

InnerMem = (InnerMem,ObsInnerMem,IntInnerMem)

where ObsInnerMem is a macro yielding false on every v of Event and

```
func IntInnerMem(dom v: Event): formula;
  v=Mem:[?,?,?,?,?]
end;
```

The informal description [3] also calls for the specification of a *reliable memory component* which is a variant of the memory component in which no MemFailure exceptions can be raised. We specify the reliable memory component by the compound system:

$$\mathsf{RSpec} = \mathsf{RMemSpec}(\mathsf{id}_0) \parallel \ldots \parallel \mathsf{RMemSpec}(\mathsf{id}_k) \parallel \mathsf{InnerMem}$$

where

$$\mathsf{RMemSpec}(\mathsf{id}) = \mathsf{MemSpec}(\mathsf{id}) \parallel \mathsf{Reliable}(\mathsf{id})$$

and Reliable(id) is the system with the same alphabets as MemSpec(id) and with normalized traces given by the following macro specifying that no exceptional return with process identity id raising MemFailure occurs.

```
func Reliable(string $\alpha$: Trace; dom id: Ident; dom vis: Visible): formula;
  $\neg$ $\exists$pos t: $\alpha$.($\alpha$(t)=Return:[?,exception,MemFailure,id?,vis?])
end;
```

That is, γ is a normalized trace of Reliable(id) if and only if Reliable(γ,id,observable) is true.

Below, when we say that have proven a formula F(γ) by feeding it to our tool we mean that we have fed a file consisting of all the type declaration for fixed k, m, n and the macro definitions given above followed by:

```
string $\gamma$: Trace;
F($\gamma$)
```

to our tool. In all executions we have $k = m = n = 1$, that is, we have two process identities, two locations and two values. Note that the reason for restricting to two of each is not reflected in the simple verification problems posed in problem 1 but rather by those of problem 3 below.

Problem 1

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

These are defined by Spec and RSpec, respectively.

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

We prove that:

$$\mathsf{RSpec}(\gamma) \;\Rightarrow\; \mathsf{Spec}(\gamma) \tag{5}$$

is a tautology by feeding the formula to our tool.

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

We first define the following macro stating that any return occurring is exceptional and raises a MemFailure exception.

```
func NothingButMemFailure(string $\alpha$: Trace): formula;
  $\forall$ dom retErr: RetErr; dom flg: Flag; pos t: $\alpha$.
      ($\alpha$(t)=Return:[?,flg?,retErr?,id?,vis?]  $\Rightarrow$  flg=exception $\wedge$ retErr=MemFailure)
end;
```

Then we prove that:

$$\mathsf{Spec}(\gamma) \wedge \mathsf{NothingButMemFailure}(\gamma) \;\Rightarrow\; \mathsf{Spec}(\gamma) \qquad (6)$$

is a tautology by running our tool. This seems reasonable for two reasons. First, there is nothing in the informal description specifying otherwise. Second, from a practical point of view disallowing such an implementation would mean disallowing an implementation involving an inner memory that could be physically destroyed or removed.

7 Implementing the memory

We now turn to the implementation of the memory component using an RPC component.

7.1 The RPC component

The problem [3] calls for a specification of an RPC component that interfaces with two components, a *sender* at a local site and a *receiver* at a remote site. Its purpose is to forward procedure calls from the local to the remote site, and to forward back the returns.

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.

We thus declare the domains:

```
type Procs    = ReadProc,WriteProc;
type NumArgs = n1,n2;
```

of procedure names Procs and of possible numbers of arguments NumArgs. As for elements of MemLocs and MemVals, we adopt the convention that being an element of Proc is a “semantic” restriction, and cannot be imposed solely by syntactic restrictions on the types of arguments. Therefore we declare:

```
type TProc = Procs & Tag;
```

The idea is that a remote procedure call passes arguments of type TProc whose first component denotes a semantically correct element of Procs if and only if the value of the Tag component is ok. The mapping ArgNum is specified by the macro:

```
func ArgNum(dom n: NumArgs; dom proc: TProc): formula;
  proc↓Procs=ReadProc ⇒ n=n1;
  proc↓Procs=WriteProc ⇒ n=n2
end;
```

where we use the Fido notation ↓ to access a field in a record. That is, proc↓Procs denotes the Procs field in the record denoted by proc.

In the informal description [3], a remote call procedure is described as:

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
Exception	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*.*

We declare the domains:

```
type Args    = Loc & Value;
type RpcErr = RPCFailure,BadCall | RetErr;
```

of argument lists and of possible exceptions raised by exceptional return errors, respectively. (Note that we restrict ourselves to lists of length at most two) In our specification, a remote procedure is called by issuing a RemoteCall event of the type:

```
type RemoteCall = TProc & NumArgs & Args & Ident;
```

A RemoteCall event takes as first parameter an element of TProc that might not be a "semantically" correct element of Procs and as second parameter an element of NumArgs denoting the length of the list from Args carried by the third parameter. The last parameter is a process identity from Ident. The return of a remote procedure is an RpcReturn event given by the declaration:

```
type RpcReturn = Value & Flag & RpcErr & Ident;
```

The first parameter is the value returned. The second indicates whether the return is normal or exceptional. In case, it is exceptional the third parameter is an element of the domain RetErr. The last parameter carries a process identity from Ident. Hence, the universe for our systems is given by:

```
type Event = Mem | Read | Write | Return | RemoteCall | RpcReturn | τ;
```

and traces (strings) over the universe by:

```
type Trace = Event(next: Trace) | empty;
```

We specify the RPC component RPC by the compound system:

$$\mathsf{RPC} = \mathsf{RPC}(\mathsf{id}_0) \parallel \ldots \parallel \mathsf{RPC}(\mathsf{id}_k)$$

defined by composing the systems RPC(id).

For a fixed process identity id in Ident, the system RPC(id) with no observable events and internal events Mem:[?,?,?,?,id], Read:[?,id,internal], Write:[?,id,internal], Return:[?,?,?,id,internal], RemoteCall:[?,?,?,id] and RpcReturn:[?,?,?,id] specifies the allowed behaviors of RPC procedure calls involving the process with identity id. The normalized traces of RPC(id) are defined by the macro:

```
func RPC(string $\alpha$: Trace; dom id: Ident): formula;
  RemoteCallAndReturnAlternates($\alpha$,id);
  RPCBehavior($\alpha$,id);
  WellTypedRemoteCall($\alpha$,id);
  OnlyInternsInRemoteCalls($\alpha$,id)
end;
```

That is, γ is a normalized trace of RPC(id) if and only if RPC(γ,id) is true. The system RPC(id) is then given by the triple:

$$\mathsf{RPC(id)} = (\mathsf{RPC(id)},\mathsf{ObsRPC(id)},\mathsf{IntRPC(id)})$$

where ObsRPC(id) is a macro that yields false on every v of Event and

```
func IntRPC(dom v: Event; dom id: Ident): formula;
  v=Mem:[rd,?,?,?,id?] $\vee$
  v=Read:[?,id,internal] $\vee$ v=Write:[?,id,internal] $\vee$  v=Return:[?,?,?,id,internal] $\vee$
  v=RemoteCall:[?,?,?,id] $\vee$ v=RpcReturn:[?,?,?,id] $\vee$
end;
```

The macro RPC is defined as the conjunction of four clauses each of which except for the last one describes properties explicitly specified in [3]. The last clause OnlyInternsInRemoteCalls specifies that any of the events Read:[?,id,internal], Write:[?,id,internal] and Return:[?,?,?,id,internal] only occurs during the handling of RPC calls. It seems clear that the specifier did not mean for read and write procedure calls on the remote site to happen without being triggered by some remote procedure call. But, the requirement is not made explicit in [3]. The first clause, RemoteCallAndReturnAlternates specifies as required in [3] that remote procedure calls are blocking in the sense that a process stops after issuing a call and waits for the corresponding return to occur. Hence, there may be multiple outstanding remote calls but not more than one triggered by the same process. Both clauses are straightforwardly defined in Fido.

For convenience, we define the following macro specifying an RPC operation by associating a RemoteCall with the corresponding RpcReturn.

```
func RpcOpr(string $\alpha$: Trace; pos $t_1$,$t_2$: $\alpha$;
        dom call,return: Event; dom id: Ident): formula;
  $t_1$<$t_2$ $\wedge$ $\alpha(t_1)$=call $\wedge$ $\alpha(t_2)$=return;
  $\neg$Between($\alpha$,$t_1$,$t_2$,RemoteCall:[?,?,?,id?]);
  $\neg$Between($\alpha$,$t_1$,$t_2$,RpcReturn:[?,?,?,id?])
end;
```

The second clause is a fairly direct transcription of the quoted lines above (excluding the last line):

```
func WellTypedRemoteCall(string $\alpha$: Trace; dom id: Ident): formula;
  $\forall$ dom proc: TProc; dom num: NumArgs;
           dom flg: Flag; dom rpcErr: RpcErr; pos $t_1$,$t_2$: $\alpha$.
   RpcOpr($\alpha$,$t_1$,$t_2$,RemoteCall:[proc?,num?,?,id?],RpcReturn:[?,flg?,rpcErr?,id?],id)
   $\Rightarrow$
   flg=normal $\Rightarrow$ proc$\downarrow$Tag=ok;ArgNum(num,proc);
   flg=exception;rpcErr=BadCall $\Leftrightarrow$ $\neg$(proc$\downarrow$Tag=ok;ArgNum(num,proc))
end;
```

stating the relationship between the parameters of a remote call and the corresponding return. The third clause specifies the properties described by:

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.*

This description is translated into the macro:

```
func RPCBehavior(string α: Trace; dom id: Ident): formula;
  ∀ dom proc: TProc; dom num: NumArgs; dom lst: Args; dom val: Value;
    dom flg: Flag; dom rpcErr: RpcErr; pos t₁,t₂: α.
     RpcOpr(α,t₁,t₂,RemoteCall:[proc?,num?,lst?,id?],RpcReturn:[val?,flg?,rpcErr?,id?],id)
     ⇒
     ABadCall(α,t₁,t₂proc,num,flg,rpcErr) ∨
     OneSuccessfulRpcCall(α,t₁,t₂,proc,lst,val,flg,rpcErr,id) ∨
     OneUnSuccessfulRpcCall(α,t₁,t₂,proc,lst,val,flg,rpcErr,id) ∨
     NoCallOfAnyProcedure(α,t₁,t₂,flg,rpcErr,id)
end;
```

where

```
func ABadCall(string α: Trace; pos t₁,t₂: α;dom proc: TProc;
          dom num: NumArgs; dom flg: Flag; dom rpcErr: RpcErr): formula;
  (¬proc↓ProcTag=Procs ∨ ¬ArgNum(num,proc)) ∧
  rpcErr=BadCall ∧ flg=exception ∧
  ¬Between(α,t₁,t₂,Read:[?,id?,internal]) ∧
  ¬Between(α,t₁,t₂,Write:[?,?,id?,internal]) ∧
  ¬Between(α,t₁,t₂,Return:[?,?,?,id?,internal])
end;

func OneSuccessfulRpcCall(string α: Trace; pos t₁: α; pos t₂: α;
            dom proc: TProc; dom lst: Args; dom val: Value;
            dom flg: Flag; dom rpcErr: RpcErr; dom id: Ident): formula;
  ∃ dom retErr: RetErr.
     ExactlyOneProcCallBetween(α,t₁,t₂,proc,lst↓Loc,lst↓Value,val,flg,retErr,id);
     flg=exception ⇒ (retErr=BadArg ⇔ rpcErr=BadArg;
                      retErr=MemFailure ⇔ rpcErr=MemFailure)
end;

func OneUnSuccessfulRpcCall(string α: Trace; pos t₁: α; pos t₂: α;
            dom proc: TProc; dom lst: Args; dom val: Value;
            dom flg: Flag; dom rpcErr: RpcErr; dom id: Ident): formula;
  flg=exception;rpcErr=RPCFailure;
  ∃ dom val_1: Value; dom flg_1: Flag; dom err: RetErr.
      ExactlyOneProcCallBetween(α,t_1,t_2,proc,lst↓Loc,lst↓Value,val_1,flg_1,err,id);
end;

func NoCallOfAnyProcedure(string α: Trace; pos t_1: α; pos t_2: α;
            dom flg: Flag; dom rpcErr: RpcErr; dom id: Ident): formula;
  flg=exception ∧ rpcErr=RPCFailure ∧
  ¬Between(α,t_1,t_2,Read:[?,id?,internal]) ∧
  ¬Between(α,t_1,t_2,Write:[?,?,id?,internal]) ∧
  ¬Between(α,t_1,t_2,Return:[?,?,?,id?,internal])
end;
```

The macro ExactlyOneProcCallBetween specifies that exactly one call of procedure proc with parameters l,v,flg and retErr occurred between t_1 and t_2, and no other internal procedure call occurred. Note that macro ABadCall additionally to the description specifies that no internal procedure call occurs.

Problem 2

Write a formal specification of the RPC component.

The RPC component is specified by the system RPC.

7.2 The implementation

A Memory component is implemented by combining an RPC component with a reliable memory component. A read or write call is forwarded to the reliable memory by issuing the appropriate call to the RPC component and the return from the RPC component is forwarded back to the caller.

We specify the implementation of the memory component Impl by the compound system:

$$\mathsf{Impl} = \mathsf{MemImpl}(\mathsf{id}_0) \parallel \ldots \parallel \mathsf{MemImpl}(\mathsf{id}_k) \parallel \mathsf{InnerMem}$$

defined by composing the systems MemImpl(id) specifying the allowed read and write procedures for fixed process identities id. Each of the systems MemImpl(id) are themselves compound systems:

$$\mathsf{MemImpl}(\mathsf{id}) = \mathsf{Clerk}(\mathsf{id}) \parallel \mathsf{RPC}(\mathsf{id}) \parallel \mathsf{IRMemSpec}(\mathsf{id})$$

For a fixed process identity id in Ident, the system Clerk(id) with observable events Read:[?,id,observable], Write:[?,id,observable] and Return:[?,?,?,id,observable], and internal events Mem:[?,?,?,?,id], Read:[?,id,internal], Write:[?,id,internal], Return:[?,?,?,id,internal], RemoteCall:[?,?,?,id] and RpcReturn:[?,?,?,id] specifies the allowed behaviors of read and write procedure calls involving the process with identity id. That is, it specifies how a local procedure call is forwarded to a remote procedure call and how the return of a remote procedure call is forwarded back as the return of the procedure call. The normalized traces of Clerk(id) are defined by the macro:

```
func Clerk(string $\alpha$: Trace; dom id: Ident): formula;
   BlockingCalls($\alpha$,id,observable);
   RPCReadStub($\alpha$,id);
   RPCWriteStub($\alpha$,id);
   RPCReturnStub($\alpha$,id);
   RetryOnlyOnRPCFailure($\alpha$,id);
   RpcOnlyInObsCall($\alpha$,id)
end;
```

That is, γ is a normalized trace of Clerk(id) if and only if Clerk(γ,id) is true. The system Clerk(id) is then given by the triple:

$$\text{Clerk(id)} = \text{(Clerk(id),ObsClerk(id),IntClerk(id))}$$

where ObsClerk(id) and IntClerk(id) are the obvious macros.

The second, third, fourth and fifth clauses of Clerk(id) are fairly direct translations of the informal description [3].

A Read *or* Write *call is forwarded to the Reliable Memory by issuing the appropriate call to the RPC component.*

```
func RPCReadStub(string $\alpha$: Trace; dom id: Ident): formula;
   $\forall$ dom l: Loc; pos $t_1$,$t_2$: $\alpha$.
      (Opr($\alpha$,$t_1$,$t_2$, Read:[l?,id?,observable],Return:[?,?,?,id?,observable],id,observable)
     $\Rightarrow$
      $\exists$ pos $t_c$,$t_r$: $\alpha$.
       ($t_1$<$t_c$; $t_r$<$t_2$;
       RpcOpr($\alpha$,$t_c$,$t_r$,RemoteCall:[[ReadProc,ok],n1,[l?,?],id?],RpcReturn:[?,?,?,id?],id)))
end;
```

The macro RPCWriteStub is similar.

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.)*

```
func RPCReturnStub(string $\alpha$: Trace; dom id: Ident): formula;
   $\forall$ dom $val_1$: Value; dom flg: Flag; dom retErr: RetErr; pos $t_1$: $\alpha$.
      $\alpha$($t_1$)=Return:[$val_1$?,flg?,retErr?,id?,observable]
     $\Rightarrow$
      $\exists$ dom $val_2$: Value; dom rpcErr: RpcErr; pos $t_0$: $\alpha$.
         $t_0$<$t_1$; $\alpha$($t_0$)=RpcReturn:[$val_2$?,flg?,rpcErr?,id?];
         $\neg$Between($\alpha$,$t_0$,$t_1$,RpcReturn:[?,?,?,id?]);
         flg=normal $\Rightarrow$ $val_1$=$val_2$;
         (flg=exception;rpcErr=RPCFailure) $\Rightarrow$ (retErr=MemFailure;
         (flg=exception;$\neg$rpcErr=RPCFailure) $\Rightarrow$ (retErr=BadArg $\Leftrightarrow$ rpcErr=BadArg;
                                               retErr=MemFailure $\Leftrightarrow$ rpcErr=MemFailure)
end;
```

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

```
func RetryOnlyOnRPCFailure(string $\alpha$: Trace; dom id: Ident): formula;
  $\forall$ pos t$_1$,t$_2$: $\alpha$.
      t$_1$<t$_2$;
      $\alpha$(t$_1$)=RemoteCall:[?,?,?,id?];
      $\alpha$(t$_2$)=RemoteCall:[?,?,?,id?];
      $\neg$Between($\alpha$,t$_1$,t$_2$,Read:[?,id?,observable]) $\wedge$
      $\neg$Between($\alpha$,t$_1$,t$_2$,Write:[?,?,id?,observable]) $\wedge$
      $\neg$Between($\alpha$,t$_1$,t$_2$,Return:[?,?,?,id?,observable])
    $\Rightarrow$
      $\exists$ pos t: $\alpha$. t$_1$<t;t<t$_2$; $\alpha$(t)=RpcReturn:[?,exception,RPCFailure,id?]
end;
```

The last clause, RpcOnlyInObsCall(α,id) specifies that a remote procedure call only occurs as the forwarding of an observable procedure call.

The systems IRMemSpec(id) specify a reliable memory with no observable events and internal events Mem:[?,?,?,?,id], Read:[?,id,internal], Write:[?,id,internal] and Return:[?,?,?,id,internal]:

$$\text{IRMemSpec(id)} = \text{IMemSpec(id)} \parallel \text{IReliable(id)}$$

where IReliable(id) are the systems with the same alphabets as IMemSpec(id) and with normalized traces given by Reliable(id,internal), and where IMemSpec(id) are defined by composition:

$$\text{IMemSpec(id)} = \text{IReadSpec(id)} \parallel \text{IWriteSpec(id)}$$

of the systems:

$$\text{IReadSpec(id)} = \text{(ReadSpec(id,internal),ObsReadSpec(id,internal),IntReadSpec(id))}$$

and the similarly defined systems IWriteSpec(id).

Problem 3

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

The implementation is specified by the system Impl. We devote the next section to proving the correctness of the implementation.

8 Verifying the implementation

We want to verify that the system Impl is an implementation of the system Spec. First, we check that the systems are comparable by running the proper instantiation of formula (1).

The obvious way to attempt verifying that the implementation is correct is to check if the formula:

$$\text{MemImpl}(\gamma,\text{id}_0) \Rightarrow \text{MemSpec}(\gamma,\text{id}_0) \quad (7)$$

holds. This is however not the case. Feeding it to the Mona tool results in the following counterexample of length 13:

```
Read:[[l_1,ok],id_0,observable]
RemoteCall:[[ReadProc,ok],n_1,[[l_1,ok],?],id_0]
Read:[[l_1,ok],id_0,internal]
Mem:[rd,l_1,v_1,normal,id_0]
Return:[[v_1,ok],normal,?,id_0,internal]
RpcReturn:[[initVal,?],exception,RPCFailure,id_0]
RemoteCall:[[ReadProc,ok],n_1,[[l_1,ok],?],id_0]
Read:[[l_1,ok],id_0,internal]
Mem:[rd,l_1,v_1,normal,id_0]
Return:[[v_1,ok],normal,?,id_0,internal]
RpcReturn:[[v_1,ok],normal,?,id_0]
Return:[[v_1,ok],normal,?,id_0,observable]
empty
```

where we have left out most of the typing information. The counterexample tells us that a successful read operation of the implementation may contain two RPC procedure calls each triggering an atomic read whereas such a read operation is not allowed by the specification. Hence, the counterexample reflects that whereas the specification requires a successful read to contain exactly one atomic read the implementation of the memory allows more than one.

An atomic read is however an internal event and fortunately, we can follow our method explained in Section 4.

To avoid explicitly building the compound system Impl(γ) of the implementation, we apply the proof rule of Theorem 7.

First, we check and see that the systems MemImpl(γ,id) $\|$ InnerMem(γ) and MemSpec(γ,id) $\|$ InnerMem(γ) for id = id$_0$,id$_1$ are comparable by running the proper instantiations of formula (1). Let Obs denote a macro defining their common alphabet of observable events and note that the internal events are defined by IntMemImpl(id) and IntMemSpec(id), respectively. Let

```
func Observe(string $\alpha$: Trace; string $\beta$: $\alpha$; dom id: Ident): formula;
  $\forall$pos t: $\alpha$.(Obs($\alpha$(t),id) $\vee$ Obs($\beta$(t),id)) $\Rightarrow$ $\alpha$(t)= $\beta$(t)
end;
```

and let

```
func R(string $\alpha$: Trace; string $\beta$: $\alpha$; dom id: Ident): formula;
  Observe($\alpha$,$\beta$,id); MemSpec($\beta$,id);InnerMem($\beta$)
end;
```

We then prove that:

$$\text{(MemImpl}(\gamma,\text{id});\text{InnerMem}(\gamma)) \Rightarrow \exists \textbf{string } \beta\text{: } \gamma.\text{R}(\gamma,\beta,\text{id}) \tag{8}$$

is a tautology (for id = id$_0$,id$_1$; the formulas are symmetric) using our tool and conclude by Proposition 6 and Theorem 3 that the system MemImpl(γ,id) $\|$ InnerMem(γ) implements MemSpec(γ,id) $\|$ InnerMem(γ) for id = id$_0$,id$_1$.

As discussed in Section 4, the compatibility requirement of Theorem 7 amounts to checking the formula (4). However, the Mona tool can not handle the state explosion caused by the existential quantification on the right hand side of the implication. Intuitively, the existential quantification guesses the internal behavior of the trace β needed to match the observable behavior of the trace γ. We can however help guessing by constraining further for each trace γ of the implementation the possible choices of matching traces β of the specification. To do this we formulate more precise (smaller) trace abstractions based on adding information of the relation between the internal behavior on the implementation and specification level.

In particular, we formalize the intuition we gained from the counterexample above that between a successful read call and the corresponding return on the implementation level exactly the last atomic read should be matched by an atomic read on the specification level. This is formalized by the macro:

```
func Map1(string α: Trace; string β: α; dom id: Ident): formula;
  ∀pos t1,t2: α.
    Opr(α,t1,t2,Read:[?,id?,observable],Return:[?,normal,?,id?,observable],id,observable)
 ⇒
    ∃pos t: α.
     t1<t;t<t2;
     α(t)=Mem:[rd,?,?,?,id?];
     α(t)=β(t);
     ¬Between(β,t1,t,Mem:[rd,?,?,?,id?]);
     ¬Between(β,t,t2,Mem:[rd,?,?,?,id?]);
     ¬Between(α,t,t2,Mem:[rd,?,?,?,id?])
end;
```

Also, we define the macro Map$_2$ specifying that an atomic read on the implementation level is matched either by the same atomic read or by a τ on the specification level:

```
func Map2(string α: Trace; string β: α; dom id: Ident): formula;
  ∀pos t: α.α(t)=Mem:[rd,?,?,?,id?] ⇒ (α(t)=β(t) ∨ β(t)=τ)
end;
```

and the macro Map$_3$ specifying that any internal event but an atomic read on the implementation level is matched by the same atomic read on the specification level and conversely, that any internal event on the specification level is matched by the same event on the implementation level:

```
func Map3(string α: Trace; string β: α; dom id: Ident): formula;
  ∀pos t: α.
     (IntMemImpl(α(t),id) ∧ ¬α(t)=Mem:[rd,?,?,?,id?]) ∨ IntMemSpec(β(t),id)
   ⇒
     α(t)=β(t)
end
```

We sum up the requirements in the macro:

```
func C(string α: Trace; string β: α): formula;
  Map1(α,β,id0); Map2(α,β,id0); Map3(α,β,id0);
  Map1(α,β,id1); Map2(α,β,id1); Map3(α,β,id1)
end;
```

We prove using our tool that:

$$\mathsf{MemImpl}(\gamma,\mathsf{id}_0);\mathsf{InnerMem}(\gamma) \;\Rightarrow \exists\textbf{string}\ \beta\colon\ \gamma.(\mathsf{C}(\gamma,\beta)) \wedge \mathsf{R}(\gamma,\beta,\mathsf{id}_0)) \tag{9}$$

is a tautology (for $\mathsf{id} = \mathsf{id}_0,\mathsf{id}_1$; the formulas are symmetric) and conclude by Proposition 6 that $\mathsf{C} \cap \mathsf{R}(\mathsf{id})$ is a trace abstraction from the system $\mathsf{MemImpl}(\gamma,\mathsf{id}) \parallel \mathsf{InnerMem}(\gamma)$ to the system $\mathsf{MemSpec}(\gamma,\mathsf{id}) \parallel \mathsf{InnerMem}(\gamma)$ for $\mathsf{id} = \mathsf{id}_0,\mathsf{id}_1$. Finally, by running our tool we prove that the formula:

$$\begin{array}{l} \quad \exists\textbf{string}\ \beta_0\colon\ \gamma.(\mathsf{C}(\gamma,\beta_0) \wedge \mathsf{R}(\gamma,\beta_0,\mathsf{id}_0)) \wedge\ \exists\textbf{string}\ \beta_1\colon\ \gamma.(\mathsf{C}(\gamma,\beta_1) \wedge \mathsf{R}(\gamma,\beta_1,\mathsf{id}_1)) \\ \Rightarrow \\ \quad \exists\textbf{string}\ \beta\colon\ \gamma.\ (\mathsf{C}(\gamma,\beta) \wedge \mathsf{R}(\gamma,\beta,\mathsf{id}_0) \wedge \mathsf{R}(\gamma,\beta,\mathsf{id}_1)) \end{array} \tag{10}$$

is a tautology and hence verify the compatibility requirement of Theorem 7 and conclude that $\mathsf{Impl}(\gamma)$ implements $\mathsf{Spec}(\gamma)$.

An alternative reaction to the failure of proving (7) is to claim to have found an error in the informal description and change the description such that it allows the behavior described by the counterexample. In our formal specification, this would amount to simply change the macro CheckSuccessfulRead to require that at least one atomic read occurs instead of exactly one. Hence modified, we prove using our tool that the formula (7) is a tautology. Likewise, we prove the symmetric formula with id_0 replaced for id_1 and conclude by propositional logic that:

$$\begin{array}{l} \quad \mathsf{MemImpl}(\gamma,\mathsf{id}_0);\mathsf{MemImpl}(\gamma,\mathsf{id}_1);\mathsf{InnerMem}(\gamma) \\ \Rightarrow \\ \quad \mathsf{MemSpec}(\gamma,\mathsf{id}_0);\mathsf{MemSpec}(\gamma,\mathsf{id}_1);\mathsf{InnerMem}(\gamma) \end{array} \tag{11}$$

and therefore by definition that:

$$\mathsf{Impl}(\gamma) \Rightarrow \mathsf{Spec}(\gamma)$$

Note that when dealing with automatic verification, the difference between the two solutions may be significant since opposed to the second the first involves the projecting out of internal behavior and hence a potential exponential blow-up in the size of the underlying automatas.

The full solution is written in 11 pages of Fido code. All the formulas (5), (6), (7), (8), (9) and (10) are decided within minutes. The largest Fido formulas specify M2L formulas of size more than 10^5 characters. During processing the Mona tool handles formulas with more than 32 free variables corresponding to deterministic automatas with alphabets of size 2^{32}. The proofs of (8), (9) and (10) required user intervention in terms of explicit orderings of the BDD variables - merging the variables encoding the traces compared.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] M. Abadi and L. Lamport. Conjoining specifications. Technical Report Report 118, Digital Equipment Corporation, Systems Research Center, 1993.

[3] M. Broy and L. Lamport. Specification problem, 1994. A case study for the Dagstuhl Seminar 9439.

[4] M. Broy and L. Lamport, editors. *The RPC-Memory Specification Problem.* Springer-Verlag, 1996. Lecture Notes in Computer Science, To appear.

[5] E. M. Clark, I.A. Browne, and R.P Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. Arnold, editor, *CAAP, LNCS 431*, pages 103–116, 1990.

[6] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, jan 1993.

[7] Emerson E.A. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. MIT Press/Elsevier, 1990.

[8] U. Engberg, Grønning P., and Lamport L. Mechanical verification of concurrent systems with tla. In *Computer Aided Verification, CAV '92.* Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 663.

[9] Urban Engberg. Reasoning in temporal logic of actions. Ph.D. Thesis, 1996.

[10] Z. Har'El and R.P. Kurshan. Software for analytical development of communications protocols. Technical report, AT&T Technical Journal, 1990.

[11] J.G. Henriksen, O.J.L. Jensen, M.E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A.B. Sandholm. Mona: Monadic second-order logic in practice. In U.H. Engberg, K.G. Larsen, and A. Skou, editors, *Procedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 58–73, 1995. BRICS Notes Series NS-95-2.

[12] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[13] Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification and simulation and refinement. In *A Decade of Concurrency*, pages 273–346. ACM, Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 803, Proceedings of the REX School/Symposium, Noordwijkerhout, The Netherlands, June 1993.

[14] N. Klarlund, M. Nielsen, and K. Sunesen. Automated logical verification based on trace abstractions. In *Proc. Fifteenth ACM Symp. on Princ. of Distributed Computing (PODC).* ACM, 1996.

[15] N. Klarlund and F.B. Schneider. Proving nondeterministically specified safety properties using progress measures. *Information and Computation*, 107(1):151–170, 1993.

[16] N. Klarlund and M.I. Schwartzbach. Logical programming for regular trees. In preparation, 1996.

[17] R. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton Univ. Press, 1994.

[18] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[19] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[20] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. Sixth Symp. on the Principles of Distributed Computing*, pages 137–151. ACM, 1987.

[21] N. Lynch and F. W. Vaandrager. Forward and backward simulations – part i: untimed systems. Technical Report CS-R9313, Centrum voor Wiskunde en Informatica, CWI, Computer Science/Department of Software Technology, 1993.

[22] Z. Manna and et al. STeP: The stanford temporal prover. In *Theory and Practice of Software Development (TAPSOFT).* Springer-Verlag, 1995. Lecture Notes in Computer Science, Vol. 915.

[23] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1991.

[24] K. L. McMillan. *Symbolic Model Checking.* PhD thesis, Carnegie Mellon University, 1993.

[25] A.P. Sistla. On verifying that a concurrent program satisfies a nondeterministic specification. *Information Processing Letters*, 32(1):17–24, July 1989.