Technical Report

UCAM-CL-TR-180
ISSN 1476-2986

Number 180

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Formal verification of data type refinement: Theory and practice

Tobias Nipkow

September 1989

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Formal Verification of Data Type Refinement

—

Theory and Practice*

Tobias Nipkow

University of Cambridge
Computer Laboratory
Pembroke Street
Cambridge CB2 3QG
England
tnn@cl.cam.ac.uk

Abstract. This paper develops two theories of data abstraction and refinement: one for applicative types, as they are found in functional programming languages, and one for state-based types found in imperative languages. The former are modelled by algebraic structures, the latter by automata. The automaton-theoretic model covers not just data types but distributed systems in general. Within each theory two examples of data refinement are presented and formally verified with the theorem prover Isabelle. The examples are an abstract specification and two implementations of a memory system, and a mutual exclusion algorithm.

Key words: Abstract Data Types, Data Types, Distributed Processes, Refinement, Implementation, Verification, Theorem Proving.

*Research supported by ESPRIT BRA grant 3245, Logical Frameworks.

Contents

1 Introduction

The aims of this paper are twofold: to present a theory of data types and their implementation, and to show how the correctness notions supplied by the theory can be verified using a theorem prover. In fact, we discuss two different approaches to abstract data types: an applicative and a state-based one. The emphasis is on correctness notions for data type implementations, i.e. the process of going from some high-level specification of a data type to a lower level implementation. This is also called *data refinement.* The test case for both theories are a specification and two implementations of the data type *memory.* The theorem prover Isabelle is used to verify the correctness of both implementations.

The paper is structured around the two theories of data types that are discussed. Both theories are firmly in the "behavioural" camp. This means that the world of types is divided into two: the "visible" basic types (booleans, integers, characters, ...), and user defined "hidden" abstract types (stacks, queues, buffers, ...). The distinction is that the latter cannot be input to or output of a program. The notion of implementation between data types can now be defined in terms of the visible input/output behaviour of programs that use these types.

The two theories differ in the mathematical model that is used to give meaning to the term data type. There are various ways to characterize the distinction: on the one hand we have applicative, functional, value-oriented, immutable, transformational, or algebraic, on the other hand state-based, imperative, object-oriented, mutable, reactive, or automaton-based. To a large part, the choice of models is dictated by the linguistic framework.

In a pure functional programming language, all data types must be of the first kind: anything is just a value, can be passed around, and can be duplicated at random. There is no distinction between basic and user defined abstract types, apart from input/output restrictions on the latter. Whereas most approaches to algebraic data types assume a deterministic world, i.e. all operations on a data type are functions, this paper develops a theory of nondeterministic data types and their implementations. The underlying mathematical model is a relational generalization of algebras. In order to develop a theory of implementations, we need to say how a data type's behaviour is observed, i.e. what the programs that use it look like. This task is complicated by the presence of nondeterminism and by our desire to design a theory of implementations that is not too dependent on a particular observation language.

The second kind of data type theory, termed state-based above, is associated with imperative programming languages. They permit the definition of, in CLU [21] terminology, "mutable" data types. Objects of these types have internal states that are never passed around outside the object but are changed by side-effect when invoking access functions. This approach is almost the norm in distributed systems, where data types are often identified with processes. Once the step to a distributed system has been made, it is natural if not compulsory to deal with nondeterminism. The obvious model is that of an automaton. The advantage of this model is that it comes with a canonical notion of behaviour: the accepted language or trace set.

We will see that in the end the distinction between the two approaches burns down

to the fact that in an applicative system, values can be duplicated (copied), whereas in a state-based system there is only one copy of the state at any time. As a consequence, state-based systems have less observational power, i.e. they make less distinctions. With respect to data refinement it means that implementations in a state-based context may not be implementations in an applicative context.

To fill the theories with life, we go through some case studies of formal verifications using a theorem prover. The basis for these examples are a collection of specifications and implementations of various storage systems found in [20]. Some of the simpler examples are selected and their correctness with respect to our notions of data refinement are shown using the theorem proving system Isabelle.

The outline of the paper is as follows. After a presentation of the three versions of the data type memory in Section 2 and a brief introduction to the theorem prover Isabelle in Section 3, the main body is devoted to the exposition of the two theories of data abstraction sketched above. Section 5 develops a theory of refinement for applicative data types and verifies Lampson's two memory implementations within that theory. Section 6 looks at the verification of distributed systems in general and state-based types in particular. After a brief review of input/output automata and their notion of refinement, the formal correctness proof of a data type (Lampson's cache memory) and a distributed process (mutual exclusion) within this theory is shown. Section 7 concludes the paper with a discussion of the differences between the two theories of data types and their correctness notions.

2 The Case Studies

The examples for our case studies are drawn from a paper by Butler Lampson [20] on the specification of distributed systems. The paper uses an extension of Dijkstra's guarded command language to specify a range of different storage systems. The particular examples we have selected are the specification of the data type *memory* and two implementations using caches. To keep this paper self-contained, we give a brief introduction to Lampson's specification language, followed by the specification of the three memory systems. In sections 5 and 6 we introduce our own formalisms for data type specification and translate his "code" into our notation. The two main points that distinguish his specifications from ours are that in section 5 we view memory as an applicative rather than a state-based type, and in section 6 we are more precise with respect to concurrency and interleaving. We are intentionally vague in this section because it is only meant as an introduction to the problems. Formality is postponed to sections 5 and 6.

2.1 Extended Guarded Commands

Dijkstra's language of guarded commands [7] was extended by Greg Nelson [26] in such a way that it becomes more suitable as a specification language than the original calculus. The major innovation is the introduction of *partial* commands, that is, commands that may fail. The following informal explanation of part of Nelson's calculus is taken from [26].

Command	Operational Meaning
skip	do nothing
$P \rightarrow A$	activate command A if P holds, else fail
$A \boxtimes B$	activate A, else B if A fails
$A; B$	activate A then B
if A fi	activate A until it succeeds
do A od	activate A until it fails
$x \mid P \rightarrow A$	activate A with a new variable x, initialized such that P holds; if P holds for no value of x, fail

The last command is in fact a combination of the two basic constructs of variable introduction and guards.

The specifications in [20] use three additional language features: grouping into atomic actions, procedures, and data abstraction, neither of which is formally defined. We bypass any questions of atomicity in the initial exposition of the examples. Atomicity becomes important only in the presence of concurrency and the possibility of interference. These issues are resolved when they arise, i.e. in section 6. For similar reasons we rely on the reader's intuition regarding procedures and data abstraction.

2.2 Simple Memory

This is the specification of a simple addressable memory. The data type imports the two types of *addresses* A and *data* D. It exports the two operations *read* and *write* [1]. The *state* of the memory m is a mapping from addresses to data, written $A \xrightarrow{m} D$. It corresponds more to an array than to a function space.

$$\text{var } m\colon A \xrightarrow{m} D$$

$$\begin{array}{lcl} read(a, \text{var } d) & = & d := m[a] \\ write(a, d) & = & m[a] := d \end{array}$$

The meaning of this specification is obvious enough. Now we look at two implementations using write-back caches.

2.3 Cache Memory

A first level implementation of the simple memory adds a single cache to the state. Read and write requests are satisfied by the cache if possible, and changes are written back to main memory immediately. The *raison d'être* of a cache is its increased access speed which is achieved at the expense of capacity. As the cache can hold only a small subset of the main memory's address space, it cannot be modelled by a mapping $A \xrightarrow{m} D$. A new element $\perp$, $\perp \notin D$, is used to denote undefinedness. $D_\perp$ is equivalent to $D \cup \{\perp\}$.

$$\begin{array}{ll} \text{var} & c\colon A \xrightarrow{m} D_\perp \\ & m\colon A \xrightarrow{m} D \end{array}$$

[1] The third operation *swap* introduced in [20] has been dropped because it is not sufficiently different from the other two.

$$
\begin{array}{lcl}
read(a, \text{var } d) & = & load(a); d := m[a] \\
write(a, d) & = & \text{if } c[a] = \bot \to flush1 \boxtimes skip \text{ fi}; c[a] := d \\
\\
load(a) & = & \text{if } c[a] = \bot \to flush1; c[a] := m[a] \boxtimes skip \text{ fi} \\
flush1 & = & a \mid c[a] \neq \bot \to m[a] := c[a]; c[a] := \bot
\end{array}
$$

The implementation ensures that the number of addresses at which c is defined remains invariant. For $flush1$ to be total, we have to assume that initially c is defined for least one address. The operations *load* and $flush1$ are auxiliary.

2.4 Coherent Cache Memory

This is a more complex version of the cache memory, suitable for a multiprocessor where each processor has its own write-back cache. The processors are identified by elements from some set P of processor numbers.

$$
\begin{array}{ll}
\text{var} & c\colon P \xrightarrow{m} A \xrightarrow{m} D_\bot \\
& m\colon A \xrightarrow{m} D
\end{array}
$$

$$
\begin{array}{lcl}
read(p, a, \text{var } d) & = & load(p, a); d := c[p, a] \\
write(p, a, d) & = & \text{if } c[p, a] = \bot \to flush1(p) \boxtimes skip \text{ fi}; c[p, a] := d; distr(p, a) \\
\\
load(p, a) & = & \text{if } c[p, a] = \bot \to \\
& & \quad flush1(p); \\
& & \quad \text{if } q \mid c[q, a] \neq \bot \to c[p, a] := c[q, a] \boxtimes c[p, a] := m[a] \text{ fi} \\
& & \boxtimes skip \text{ fi} \\
distr(p, a) & = & \text{do } q \mid c[q, a] \neq \bot \wedge c[q, a] \neq c[p, a] \to c[q, a] := c[p, a] \text{ od} \\
flush1(p) & = & a \mid c[p, a] \neq \bot \to m[a] := c[p, a]; c[p, a] := \bot
\end{array}
$$

Writing $c[p, a]$ instead of $c[p][a]$ follows the usual convention of indexing multi-dimensional arrays. The formal justification is the isomorphism between $A \xrightarrow{m} B \xrightarrow{m} C$ and $A \times B \xrightarrow{m} C$.

Although it may seem we have changed the interface by adding the parameter p to *read* and *write*, this is not the case. The parameter p has only been introduced on the conceptual level. For each processor p the interface remains as specified in section 2.2. We have simply replaced indexed sets of operations $read_p$ and $write_p$ by a single one.

Note that this specification is still a long way from an implementation. In particular it needs an efficient realization of the auxiliary operation *distr*, which broadcasts a change in one cache to all other caches.

3 Isabelle

Isabelle is a generic theorem prover developed by Larry Paulson at the University of Cambridge. By supplying syntax and inference rules it can be instantiated to support particular logics. Isabelle's style of theorem proving stands in the LCF [34] tradition, i.e. it is interactive and driven by user defined tactics. For an overview of Isabelle see [35]. Isabelle's logical foundation is explored in [36].

All theorems presented in this paper were proved by rewriting related tactics and induction. An anatomy of these tactics can be found in [31]. Their application in the correctness proofs of some sorting algorithms is detailed in [32]. The proofs of the theorems in this paper are in the same style and have been omitted. Only the sequence of lemmas leading up to them is reproduced.

3.1 Isabelle as a Specification Language

In order to use Isabelle for program verification, we identify specifications with logics or extension of logics. Each extension introduces

- a set of new type (constructor) names,
- a set of new (logical or non-logical) constants, and
- a set of axioms and inference rules.

In analogy to OBJ [9] we use the following syntax:

```
Extension = Base1 + ... + Basen + SORTS ... OPS ... RULES ...
```

This means that `Extension` is the extension of the union of the theories `Base1` through `Basen` with the types listed after `SORTS`, the constants after `OPS`, and the inference rules after `RULES`. The syntax of constants is given in mixfix notation. The notation for types follows ML [13] conventions.

This OBJ-like syntax is different from but closely related to the actual syntax used in the definition of Isabelle logics. Apart from some minor matters of surface syntax, the only liberty we have taken is the inclusion of type constructors with arguments, and polymorphic constants. Both are currently not supported by Isabelle, but there are plans for such extensions, along the lines of LCF's [34] and HOL's [11] type system. The actual proofs were conducted in a single-sorted logic. It is only for the sake of presentation that we have introduced many-sortedness and polymorphism.

The starting point for all our specifications is an axiomatization of first-order logic with equality. In addition to the usual logical symbols a conditional, pairs, and triples are defined:

$$
\begin{array}{l}
\texttt{FOLE} = \texttt{SORTS}\ \mathit{form},\ (\alpha,\beta)\mathit{pair},\ (\alpha,\beta,\gamma)\mathit{triple} \\
\quad \texttt{OPS}\ \_\wedge\_, \_\vee\_, \_\Rightarrow\_, \_\Leftrightarrow\_ : \mathit{form} * \mathit{form} \to \mathit{form} \\
\qquad \_=\_ : \alpha * \alpha \to \mathit{form} \\
\qquad \vdots \\
\qquad \text{if}\_\to\_\boxtimes\_\text{fi} : \mathit{form} * \alpha * \alpha \to \alpha \\
\qquad \langle \_, \_ \rangle : \alpha * \beta \to (\alpha,\beta)\mathit{pair} \\
\qquad \langle \_, \_, \_ \rangle : \alpha * \beta * \gamma \to (\alpha,\beta,\gamma)\mathit{triple} \\
\quad \texttt{RULES} \\
\qquad \vdots \\
\qquad C(\text{if } P \to x \boxtimes y \text{ fi}) \ \Leftrightarrow\ (P \Rightarrow C(x)) \wedge (\neg P \Rightarrow C(y))
\end{array}
$$

Notice that the polymorphic type of if_→_□_fi allows it to be used for both formulae and expressions. Its defining rule uses the higher order variable C of type $\alpha \to form$ representing "contexts". Conditionals are only a notational extension of the calculus because they can be removed from any formula containing them.

The decision to use a standard first-order logic is one of convenience. An Isabelle instantiation exists, and logics for total functions are easier to reason in than those for partial functions. However, it means that any extension with partial functions leads to inconsistencies. In the Boyer-Moore system [2] this problem is dealt with by verifying formally that all new functions are total. The consistency of the extensions of FOLE presented below has been checked informally only.

4 General Remarks on Implementations

Since the main theme of this volume is refinement, it is appropriate to make some general remarks on this topic before delving into technicalities. We feel that the proper starting point for any treatment of implementations is the following intuition:

Definition 1 A component C implements a component A if and only if the behaviour of any system with component C is also a behaviour of the same system with C replaced by A [2].

This defines what we call the *implementation preorder* $C \leq A$.

Although it remains to be fixed what "components", "systems", and "behaviour" are, we think that most computer scientists would agree to this definition [3].

Despite its generality, this definition makes some tacit assumptions by identifying specifications, implementations, and components. If specifications may denote sets of components, we could define that a set of components M implements a set of components N if for all $C \in M$ there is an $A \in N$ such that $C \leq A$ in the sense of Definition 1. However, that forces us to reason about sets rather than single components. In the sequel we stick to our original definition and assume that it suffices to compare individual components. If the specification formalism ensures that the set of components denoted by a specification always has a largest element with respect to the implementation preorder, these elements can be used as representatives in a correctness proof. More precisely, given a set M with greatest component C and a set N with greatest component A, M implements N if and only if $C \leq A$.

The degree to which the observational view has determined the treatment of refinement in the fields of data types and concurrency is markedly different.

In the data type field it is mostly the case that some abstract mathematical notion such as homomorphism is taken as the definition of refinement without any justification in terms of behaviour. A notable exception is the work of Schoett [37,38], who starts from exactly the premises above. For a survey of other approaches to implementations of data types the reader should consult [38] or [29]. Definition 1 can be applied to data

[2] Read *Abstract* and *Concrete* for A and C.

[3] A possible moot point is that the implementation may display only some of the specification's behaviour. Kuiper [19] for example suggests distinguishing *allowed* from *required* nondeterminism. The second kind must be preserved by implementations.

types by identifying data types with components and programs with systems. The point is that in contrast to data types, programs come with more or less canonical notions of behaviour in the form of input/output traces. Implementations between data types are now defined in terms of the induced behaviours of programs using the data types.

In the algebra of concurrent processes, the relation $\leq$ is known as a *testing preorder* [6]. Components and systems are both identified with concurrent processes. Since processes come with well-defined notions of behaviour (e.g. in the form of traces), this leads to a notion of *observational equivalence* and *observational congruence* [17]. In automaton-based approaches like [23] or [1], the definition of refinement is based directly on the trace sets generated by the automata.

The two theories studied in this paper are firmly in the behavioural camp.

5 Applicative Data Types

5.1 Nondeterministic Data Types and Their Implementation

This section reviews the theory of nondeterministic data types (for short: data types) established in [27,29,30]. After a brief introduction of the mathematical model we have chosen for data types, we focus on the question of implementation for the rest of the section. In particular we show how the global definition of implementation given in Section 4 can be localized for particular choices of observing systems and behaviours. This means a characterization of refinement as a set-theoretic relationships between models. The important notion is that of a simulation, which is both a relational generalization of homomorphism and half a bisimulation [33].

All reasoning is on the semantic level of models and thus independent of any particular specification formalism. As discussed in Section 4, specifications and implementations are identified with models. Section 5.1.4 deals with the application to specific formalisms and the step from semantics back to syntax.

5.1.1 Models

The interface to a data type is called its *signature*. It lists the *sorts* and operations exported by the type; sorts are classified as visible or hidden.

Definition 2 *A* signature *is a triple* $\Sigma = (S, V, O)$ *where* S *is a set of* sort *names,* $V \subseteq S$ *the set of* visible *sorts, and* O *a set of* operations. *Each operation* $r \in O$ *is typed as* $r: w \to s$, *where* $w \in S^*$ *and* $s \in S$.

In the sequel assume $\Sigma = (S, V, O)$.

Data types are modelled by *structures*, which almost coincide with *multi-algebras* [12] and are closely related to structures in logic [40].

Definition 3 *A* Σ-structure A *consists of*

- *an* S*-indexed family of sets* A_s, $s \in S$, *and*
- *a relation* $r^A \subseteq A_w \times A_s$ *for each operation* $r: w \to s$ *in* O.

For $r: w \to s$ *in* O *and* $a \in A_w$ *define* $r^A(a) = \{b \in A_s \mid (a, b) \in r^A\}$.

The interpretation of a pair $(a, b) \in r^A$ is that operation r called with argument tuple a *may* return b. If $r^A(a)$ is empty, r is undefined for a, i.e. it diverges.

This model cannot express possible termination or divergence. If $r^A(a)$ is empty, r^A never terminates if applied to a, otherwise it always terminates. To get a finer distinction, we have to extend the model. One can either introduce a special element $\perp$ such that $\perp \in r^A(a)$ means that $r^A(a)$ may diverge, or an explicit *termination set* for each operation, containing the inputs for which termination is guaranteed. Both choices can be found in the literature. In the sequel we work with $\perp$ and require that $r^A(a)$ is always non-empty, i.e. contains at least $\perp$. A structure is now called *total* if $\perp$ does not occur in the range of any of its operations.

5.1.2 Three Implementation Concepts

Having fixed what the components in the sense of Section 4 are, we need a set of observers or systems to exercise the components. It should be a programming language that is general enough to be representative for a wider class of languages and simple enough to be tractable. Ideally, a kind of λ-calculus for applicative nondeterministic computation structures is required. Given such an observation language, one can ask for a characterization of the implementation relation it induces between data types. Alternatively, one can fix some preorder $\sqsubseteq$ between data types and determine requirements on a language semantics which ensure that $C \sqsubseteq A$ implies that C is an implementation of A with respect to any language meeting those requirements. If $\sqsubseteq$ is well chosen those requirements should be the semantic counterpart of the information hiding principle: they should ban all language features which permit access to the representation of an abstract data type. One can then show that particular languages meet these requirements. This is the approach taken in [27,29,30], the technical details of which are only sketched here. It requires a syntactic domain of programs *Prog*, a semantic domain *Sem* with an implementation preorder on it, and a mapping $D[\![.,.]\!]$ which takes a program and a Σ-structure and returns a denotation in *Sem*.

For the time being we do not fix *Sem* completely but assume that it is based on powerdomains [39]. Powerdomains are extensions of domains to powersets. A partial order on a domain D can be extended to a preorder on $P(D)$ in different ways, two of which give rise to important correctness concepts:

$$M \preceq_1 N \;\Leftrightarrow\; \forall m \in M\ \exists n \in N.\ m \preceq n$$
$$M \preceq_2 N \;\Leftrightarrow\; \forall m \in M\ \exists n \in N.\ n \preceq m$$

The predicates $\preceq_1$ and $\preceq_2$ are the orderings of the so called *Hoare* [16] and *Smyth*[4] [41] powerdomains respectively. We also define $\preceq_0$ to be $\subseteq$. Broy [4] associates the adjectives *loose*, *partial*, and *robust* with the orderings $\preceq_0$, $\preceq_1$ and $\preceq_2$, a terminology we adopt. We assume that *Sem* comes with either of the orderings $\preceq_i$ as the intended notion of refinement between program denotations.

Considering the carriers of a structure as trivially ordered flat domains and the operations as functions returning power sets, the latter can be compared via $\preceq_i$. This in turn extends to structures by defining $A \preceq_i B$ iff $r^A \preceq_i r^B$ for all operations r.

[4]Note that $M \preceq_2 N$ is usually written $N \preceq_2 M$.

Of the three notions, loose correctness is the strongest. It only allows reduced nondeterminism. Without going into details, let us just mention that partial correctness corresponds to "safety" and robust correctness to "liveness" in the language of distributed systems.

The above correctness notions carry over to data types as follows. We write $C \leq_i A$ if $D[\![p, C]\!] \preceq_i D[\![p, A]\!]$ holds for al programs p. In that case we call C a loose (partial, robust) implementation of A. What we are interested in is to characterize $\leq_i$ without any reference to observing programs, purely as a set-theoretic relation between data types. Our main tools for that purpose are the so called simulations.

Definition 4 *Let C, A be two Σ-structures and let $\sqsubseteq$ be an S-sorted relation $\sqsubseteq_s \subseteq C_s \times A_s$ such that $\sqsubseteq_v$ is the identity for all visible sorts $v \in V$ and $\bot \sqsubseteq \bot$ is the only pair in $\sqsubseteq$ containing $\bot$. For $w \in S^*$ let $\sqsubseteq_w$ be the componentwise extension of $\sqsubseteq_s$.*

$\sqsubseteq$ is called a partial simulation *iff for all operations $r : w \to s$ and all a, c and $c' \neq \bot$ we have*

$$c \sqsubseteq_w a \wedge (c, c') \in r^A \;\Rightarrow\; \exists a'.\ (a, a') \in r^A \wedge c' \sqsubseteq_s a'$$

$\sqsubseteq$ is called a (loose) simulation *iff for all operations $r : w \to s$ and all a, c and c' we have*

$$c \sqsubseteq_w a \wedge (c, c') \in r^A \;\Rightarrow\; \exists a'.\ (a, a') \in r^A \wedge c' \sqsubseteq_s a'$$

$\sqsubseteq$ is called a robust simulation *iff for all operations $r : w \to s$ and all a, c and c' we have*

$$c \sqsubseteq_w a \wedge (a, \bot) \notin r^A \;\Rightarrow\; (c, \bot) \notin r^C \wedge ((c, c') \in r^C \Rightarrow \exists a'.\ (a, a') \in r^A \wedge c' \sqsubseteq_s a')$$

We write $C \sqsubseteq_i A$ to indicate that there is a loose ($i = 0$), partial ($i = 1$) or robust ($i = 2$) simulation between C and A. Note that for total structures C and A, all three simulations coincide.

Simulations are generalizations of the correctness notions $\preceq_i$: $C \preceq_i A \;\Rightarrow\; C \sqsubseteq_i A$. The weaker relation $\preceq_i$ can only relate structures with identical carriers, whereas $\sqsubseteq_i$ can relate arbitrary structures. This shows that simulations have a twofold task: to relate different carriers and to guarantee the desired correctness notion.

It remains to be seen what simulations have to do with implementations induced by programs. In particular we are interested in the notions of

Soundness : does $C \sqsubseteq_i A$ imply $C \leq_i A$?

Completeness : does $C \leq_i A$ imply $C \sqsubseteq_i A$?

In [29,30] we obtained some rather abstract criteria for soundness based on the decomposition of simulations into simpler relations like homomorphisms and $\preceq$. We showed that $\sqsubseteq_i$ is sound provided that

- D is insensitive to "junk": if B is a substructure of A, i.e. B contains less unreachable elements (junk) than A, then $D[\![p, B]\!] = D[\![p, A]\!]$ should hold.
- D reflects homomorphisms on hidden values accurately: if B is a homomorphic image of A then $D[\![p, B]\!]$ should be a homomorphic image of $D[\![p, A]\!]$.

- D is monotone w.r.t. $\preceq_i$: $B \preceq_i A$ should imply $D[\![p, B]\!] \preceq_i D[\![p, A]\!]$.

The first two requirements of D should be interpreted as the semantic counterpart of the information hiding principle mentioned at the beginning of this section. The third point is a trivial monotonicity requirement which languages with a denotational semantics should satisfy anyway.

In order to show that real toy languages meet these requirements, we look at a particular instantiation for *Prog*, *Sem*, and D. Without going into details (they can be found in [3,27,29,30]), let us just say that the observation language is a first-order applicative stream processing language similar to Broy's AMPL. In particular it features an angelic choice construct. We call this language L and the subset obtained by removing angelic choice L'. The semantic domains are so called *streams* with the *approximation* ordering $\preceq$. It is this ordering $\preceq$ which the $\preceq_i$ extend to powersets of streams. In this particular instance we obtain the following theorem:

Theorem 1 *Loose (partial) simulations are sound criteria for loose (partial) implementations with respect to programs over L.*

Robust simulations are sound criteria for robust implementations with respect to programs over L' but not over L.

The problem with robust implementations is that in case the specified operation diverges for a certain input, the implementation is free to do what it likes: if f^A is a function that always diverges, the function f^C which always returns 1 is a robust implementation. But a program that chooses angelically between 0 and $f(x)$ will always return 0 (and terminate!) if it uses A, whereas it may also return 1 if it uses C.

Completeness results depend very much on the expressive power of the observing programming language. For the particular example of L we have

Theorem 2 *A structure A is called* finitely nondeterministic *iff $r^A(a)$ is always finite. For the subclass of finitely nondeterministic structures, loose (partial) simulations are complete criteria for loose (partial) implementations with respect to programs over L.*

The extension to infinite nondeterminism is still open.

This shows that, modulo finite nondeterminism, loose and partial simulations exactly characterize loose and partial implementations with respect to L.

A completeness result for robust simulations holds only for a rather restricted subclass of structures. Details can be found in [29,30].

5.1.3 Homomorphisms

The basic semantic tool for connecting specifications and implementations is a relation, the simulation. An important special case is that of functional simulations, which are easily seen to be homomorphisms. Hoare [14] was one of the first to define data refinement formally using homomorphisms and many authors have since followed him [18,5]. There are two practical reasons for this:

- Functions are in general easier to handle than relations; in particular the existential quantifier in the definition of a simulation disappears. Thus the approach becomes amenable to support from term-rewriting based verification systems like LP [10].

- It is a fact of life that homomorphisms suffice for most of the verification tasks arising in practice.

It is the second point that we want to look at in more detail. To start with, it is not difficult to establish that homomorphisms are not complete, i.e. there are structures $C \sqsubseteq A$, such that there is no homomorphism from C to A. Most examples of this kind are such that there is a third structure B which is behaviourally equivalent[5] to A and a homomorphic image of C. Jones [18] classifies this as an "implementation bias" in the specification. Had B been chosen as the specification instead of A, C could have been shown to be a correct implementation via a homomorphism. Therefore the question is: does every specification have a behaviourally equivalent counterpart which is the homomorphic image of all its implementations. Technically speaking, we are interested in the existence of final objects in a class of behaviourally equivalent structures.

Summarizing the results obtained in [28] we can say that for deterministic specifications such final objects always exist, whereas in general they don't. This means that in a deterministic world homomorphisms suffice because one can always start with a fully abstract, i.e. final, specification. In the presence of nondeterminism one may be forced to work with proper simulations because fully abstract specifications no longer exist.

More precisely, the results are as follows. If we restrict ourselves to partial algebras, i.e. structures where $r^A(a)$ is always a singleton set, fully abstract specifications exist with respect to loose and robust, but not partial implementations. If we consider all structures, fully abstract specifications cease to exist for all notions of implementations discussed in this paper. Only bisimulation equivalence, which is strictly finer than all the above notions, admits fully abstract specifications.

Despite the conclusion that simulations are more general, homomorphisms turn out to be sufficient for the correctness proofs in this paper. Using a function $\varphi: C \to A$, correctness of C is depicted in Figure 1: the three solid lines imply the existence of the dashed line completing the square. The corresponding proposition is

$$(c, c') \in r^C \;\Rightarrow\; (\varphi(c), \varphi(c')) \in r^A, \tag{1}$$

which is exactly the definition of a homomorphism in [12].

5.1.4 Syntax versus Semantics

Having arrived at the actual formula that is the correctness notion in the examples to come, we want to give a brief indication how the results obtained can be applied to particular specification languages. On the one hand there are formalisms like VDM [18], Z [42], or initial or final algebra specifications [8] which associate a canonical model with each specification. Our theory is tailor-made for such formalisms. In fact, defining a notion of implementation for Z has lead to a very similar theory of data refinement [15], although with a state-based view of types.

On the other hand there are formalisms which associate a whole class of models with a specification, e.g. the loose approach to algebraic specifications [8]. Even worse,

[5] A and B are termed *behaviourally equivalent* if they have the same set of implementations.

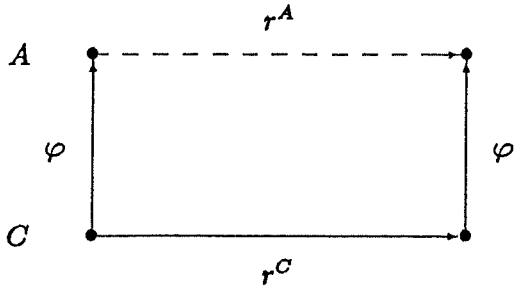

Figure 1: A Homomorphism

approaches like [24] come with no model theoretic semantics at all. Nevertheless our theory is still applicable by translating from semantics to syntax. If formal verification is to be supported mechanically, one has to abandon the realm of models in favour of purely syntactic formalisms anyway. In general one is faced with four sets of sentences B, S, I and H, describing the the basic types, the specification, the implementation, and the homomorphism between them. Proving correctness means showing that H does in fact specify a homomorphism:

$$B \cup S \cup I \cup H \vdash \Phi \tag{2}$$

where Φ is (1) above. If S and I are conservative extensions of B with disjoint vocabulary, and H is a conservative extension of $B \cup S \cup I$, there is a simple translation of (2) back into the realm of semantics: from any model of I there is a homomorphism to any other model of S. Thus any model of I implements any model of S, which is certainly sufficient, although in fact stronger than the condition given in Section 4.

This concludes the treatment of the theoretical underpinnings for the actual proofs to come.

5.2 Translation into First Order Logic

The translation of Lampson's specifications in Section 2 into predicate calculus involves both the basic data types, in this case only maps, and the operations of the defined types. For our purposes the following specification of polymorphic maps, called $\xrightarrow{m}$ in Section 2, suffices.

Map = FOLE +
SORTS $(\alpha,\beta)map$
OPS $\_[\_] : (\alpha,\beta)map * \alpha \to \beta$
$\_[\_/\_] : (\alpha,\beta)map * \beta * \alpha \to (\alpha,\beta)map$
$\_\backslash\_ : (\alpha,\beta)map * \alpha \to (\alpha,\beta)map$
$D : (\alpha,\beta)map * \alpha \to form$
RULES
$m[b/a][a'] = \text{if } a' = a \to b \boxtimes m[a'] \text{ fi}$
$a \neq a' \Rightarrow (m\backslash a)[a'] = m[a']$

$$D(m[b/a], a') \Leftrightarrow a = a' \vee D(m, a')$$
$$D(m\backslash a, a') \Leftrightarrow a \neq a' \wedge D(m, a')$$
$$m = m' \Leftrightarrow \forall a.\ m[a] = m'[a]$$

Instead of returning $\perp$ in case the map is undefined for some argument as in Section 2, we have introduced an explicit definedness predicate D. Notice also that Map has an empty initial model [8] because it lacks constants of type *map*, e.g. the empty map. This is a reflection of the fact that Lampson's memory specifications do not talk about initial states. Of course the correctness theorems we are about to prove remain valid in any extension of the current theory which fixes those details.

Translation of the guarded command text into predicate calculus involves two changes. The specification in Section 2 views the memory as a global variable. In the applicative context of this section, it becomes an additional parameter to each operation which is passed into and out of the operation. Operations are modelled by predicates, which means they are formulae, i.e. constants with result type *form*. The translation of the actual code was guided by Nelson's [26] translation from guarded commands to first-order formulae expressing the relation between pre and post states. In some places the resulting formulae were simplified slightly.

It has to be emphasized that we do not introduce a constant $\perp$, as in Section 5.1, to model nontermination. Instead we stick with the simple model of Definition 3, where divergence of operation r for input c is modelled by $r^C(c) = \{\}$. As a consequence, proposition (1) expresses only partial correctness: if r^C is empty, the proposition always holds. To prove loose correctness, the additional proposition

$$(f(c), a') \in r^A \Rightarrow \exists c'.\ (c, c') \in r^C$$

has to be verified. For simplicity we establish only partial correctness.

5.3 Simple Memory

Lampson's specification of the simple memory translates into

SM = Map +
OPS *read*: $(\alpha, \delta)map * \alpha * (\alpha, \delta)map * \delta \rightarrow form$
write: $(\alpha, \delta)map * \alpha * \delta * (\alpha, \delta)map \rightarrow form$
RULES
$read(m, a, m', d) \Leftrightarrow m' = m \wedge d = m[a]$
$write(m, a, d, m') \Leftrightarrow m' = m[d/a]$

The operations *read* and *write* have become predicates relating the memory state before (m) and after (m') the execution and the input and output parameters (a, d). Addresses, data and memory are represented by the type variables α, δ and $(\alpha, \delta)map$ respectively. The polymorphic nature of the specification expresses very clearly that (on an abstract level) memory is independent of the structure of addresses or data.

5.4 Cache Memory

The translation of Lampson's cache memory is fairly straightforward:

CM = Map +
SORTS $(\alpha,\delta)cm = ((\alpha,\delta)map,(\alpha,\delta)map)pair$
OPS $read'$: $(\alpha,\delta)cm * \alpha * (\alpha,\delta)cm * \delta \to form$
$write'$: $(\alpha,\delta)cm * \alpha * \delta * (\alpha,\delta)cm \to form$
$load$: $(\alpha,\delta)cm * \alpha * (\alpha,\delta)cm \to form$
$flush1$: $(\alpha,\delta)cm * (\alpha,\delta)cm \to form$
RULES
$read'(\langle c,m\rangle,a,\langle c',m'\rangle,d) \Leftrightarrow load(\langle c,m\rangle,a,\langle c',m'\rangle) \wedge d = c'[a]$
$write'(\langle c,m\rangle,a,d,\langle c',m'\rangle) \Leftrightarrow$ if $D(c,a) \to c' = c[d/a] \wedge m' = m$
$\boxtimes\, \exists c''.\ flush1(\langle c,m\rangle,\langle c'',m'\rangle) \wedge c' = c''[d/a]$ fi
$load(\langle c,m\rangle,a,\langle c',m'\rangle) \Leftrightarrow$ if $D(c,a) \to c' = c \wedge m' = m$
$\boxtimes\, \exists c''.\ flush1(\langle c,m\rangle,\langle c'',m'\rangle) \wedge c' = c''[m'[a]/a]$ fi
$flush1(\langle c,m\rangle,\langle c',m'\rangle) \Leftrightarrow \exists a.\ D(c,a) \wedge c' = c\backslash a \wedge m' = m[c[a]/a]$

The interface operations are now called $read'$ and $write'$ to distinguish them from those in the specification SM. This is necessary because the correctness requirement (1) talks about both of them at the same time. Correctness is shown by proving that the following function φ is a homomorphism:

I1 = SM + CM +
OPS φ: $(\alpha,\delta)map * (\alpha,\delta)map \to (\alpha,\delta)map$
RULES
$\varphi(c,m)[a] =$ if $D(c,a) \to c[a] \boxtimes m[a]$ fi

Although φ is supposed to produce a mapping, its definition does not actually say what the result mapping is. Instead, it is characterized implicitly by its behaviour w.r.t. application. Because maps are extensional (last axiom in Map), this implicit specification is sufficient.

Correctness, proved in the joint theory I1, is immediate for both *read* and *write*:

$$\begin{aligned} read'(\langle c,m\rangle,a,\langle c',m'\rangle,d) &\Rightarrow read(\varphi(c,m),a,\varphi(c',m'),d) \\ write'(\langle c,m\rangle,a,d,\langle c',m'\rangle) &\Rightarrow write'(\varphi(c,m),a,d,\varphi(c',m')) \end{aligned}$$

5.5 Coherent Cache Memory

The coherent cache memory specification in Section 2.4 translates into

CCM = Map +
SORTS $(\pi,\alpha,\delta)cache = ((\pi,\alpha)pair,\delta)map$
$(\pi,\alpha,\delta)ccm = ((\pi,\alpha,\delta)cache,(\alpha,\delta)map)pair$
OPS $read'$: $(\pi,\alpha,\delta)ccm * \pi * \alpha * (\pi,\alpha,\delta)ccm * \delta \to form$
$write'$: $(\pi,\alpha,\delta)ccm * \pi * \alpha * \delta * (\pi,\alpha,\delta)ccm \to form$
$load, ld$: $(\pi,\alpha,\delta)ccm * \pi * \alpha * (\pi,\alpha,\delta)ccm \to form$
$distr$: $(\pi,\alpha,\delta)cache * \pi * \alpha \to (\pi,\alpha,\delta)cache$
co : $(\pi,\alpha,\delta)cache \to form$
RULES
$read'(\langle c,m\rangle,p,a,\langle c',m'\rangle,d) \Leftrightarrow load(\langle c,m\rangle,p,a,\langle c',m'\rangle) \wedge d = c'[\langle p,a\rangle]$

$$
\begin{array}{l}
write'(\langle c,m\rangle,p,a,d,\langle c',m'\rangle) \Leftrightarrow \exists c''.\\
\qquad \text{if } D(c,\langle p,a\rangle) \rightarrow c'' = c \wedge m' = m \boxtimes flush1(\langle c,m\rangle,p,\langle c'',m'\rangle) \text{ fi} \wedge\\
\qquad c' = distr(c''[d/\langle p,a\rangle],p,a)\\
load(\langle c,m\rangle,p,a,\langle c',m'\rangle) \Leftrightarrow\\
\qquad \text{if } D(c,\langle p,a\rangle) \rightarrow c' = c \wedge m' = m \boxtimes ld(\langle c,m\rangle,p,a,\langle c',m'\rangle) \text{ fi}\\
ld(\langle c,m\rangle,p,a,\langle c',m'\rangle) \Leftrightarrow \exists c''.\ flush1(\langle c,m\rangle,p,\langle c'',m'\rangle) \wedge\\
\qquad \text{if } \forall q.\neg D(c'',\langle q,a\rangle) \rightarrow c' = c''[m[a]/\langle p,a\rangle]\\
\qquad \boxtimes \exists q.\ D(c'',\langle q,a\rangle) \wedge c' = c''[c''[\langle q,a\rangle]/\langle p,a\rangle] \text{ fi}\\
flush1(\langle c,m\rangle,p,\langle c',m'\rangle) \Leftrightarrow \exists a.\ D(c,\langle p,a\rangle) \wedge c' = c\backslash\langle p,a\rangle \wedge m' = m[c[\langle p,a\rangle]/a]\\
distr(c,p,a)[\langle q,b\rangle] = \text{if } b = a \rightarrow c[\langle p,a\rangle] \boxtimes c[\langle q,b\rangle] \text{ fi}\\
D(distr(c,p,a),\langle q,b\rangle) \Leftrightarrow D(c,\langle q,b\rangle)\\
co(c) \Leftrightarrow \forall a.\forall p.\forall q.\ D(c,\langle p,a\rangle) \wedge D(c,\langle q,a\rangle) \Rightarrow c[\langle p,a\rangle] = c[\langle q,a\rangle]
\end{array}
$$

The type variable π is used in places where Lampson's specification talks about the set P of *processors*. Apart from *distr*, CCM is a fairly direct translation of Lampson's imperative specification. Because the result (not the computation!) of *distr* is deterministic, *distr* has become a function rather than a relation on caches. It is defined implicitly by its behaviour w.r.t. application and definedness.

The predicate *co* specifies coherence. In order to prove that coherence is an invariant property of this system, we need the following lemmas:

$$
\begin{array}{rcl}
D(c,\langle p,a\rangle) \wedge D(c,\langle q,a\rangle) \wedge co(c) & \Rightarrow & c[\langle p,a\rangle] = c[\langle q,a\rangle]\\
co(c) & \Rightarrow & co(c\backslash\langle p,a\rangle)\\
(\forall q.\ \neg D(c,\langle q,a\rangle)) \wedge co(c) & \Rightarrow & co(c[d/\langle p,a\rangle])\\
D(c,\langle p,a\rangle) \wedge co(c) & \Rightarrow & co(c[c[\langle p,a\rangle]/\langle q,a\rangle])\\
co(c) & \Rightarrow & co(distr(c[d/\langle p,a\rangle],p,a))
\end{array}
$$

Preservation of coherence by *read'* and *write'* can now be proved in a single step.

$$co(c) \wedge read'(\langle c,m\rangle,p,a,\langle c',m'\rangle,d) \Rightarrow co(c') \qquad (3)$$

$$co(c) \wedge write'(\langle c,m\rangle,p,a,d,\langle c',m'\rangle) \Rightarrow co(c') \qquad (4)$$

Proving partial correctness of CCM w.r.t. SM involves the homomorphism φ:

```
I2 = SM + CCM +
     OPS φ: (π,α,δ)cache * (α,δ)map → (α,δ)map
     RULES
        co(c) ∧ D(c,⟨p,a⟩) ⇒ φ(c,m)[a] = c[⟨p,a⟩]
        co(c) ∧ (∀p. ¬D(c,⟨p,a⟩)) ⇒ φ(c,m)[a] = m[a]
```

Notice that removing the assumption $co(c)$ leads to an inconsistency: if both $D(c,\langle p,a\rangle)$ and $D(c,\langle q,a\rangle)$ hold, it would follow that $c[\langle p,a\rangle] = \varphi(c,m)[a] = c[\langle q,a\rangle]$, which is consistent only if $co(c)$ holds.

The following additional lemmas were required before φ could be shown to be a homomorphism.

$$co(c) \wedge co(c[d/\langle p,a\rangle]) \Rightarrow \varphi(c[d/\langle p,a\rangle],m)[b] = \text{if } b = a \rightarrow d \boxtimes \varphi(c,m)[b] \text{ fi}$$

$$
\begin{aligned}
co(c) \wedge b \neq a &\Rightarrow \varphi(c, m[d/a])[b] = \varphi(c, m)[b] \\
co(c) \wedge b \neq a &\Rightarrow \varphi(c \backslash \langle p, a \rangle, m)[b] = \varphi(c, m)[b] \\
co(c) \wedge b \neq a &\Rightarrow \varphi(distr(c[d/\langle p, a \rangle], p, a), m)[b] = \varphi(c, m)[b]
\end{aligned}
$$

The correctness statements for *read'* and *write'* differ slightly from those for the cache memory because we have to take coherence into account:

$$
\begin{aligned}
co(c) \wedge read'(\langle c, m \rangle, p, a, \langle c', m' \rangle, d) &\Rightarrow read(\varphi(c, m), a, \varphi(c', m'), d) \\
co(c) \wedge write'(\langle c, m \rangle, p, a, d, \langle c', m' \rangle) &\Rightarrow write(\varphi(c, m), a, d, \varphi(c', m'))
\end{aligned}
$$

Lemmas (3) and (4) justify assuming $co(c)$. Remember that the free occurrence of p means correctness of *read'* and *write'* is proved for every processor p.

6 State-Based Data Types

The paradigm explored in this section is that of state-based systems. Their canonical model is that of an automaton. The term data type is in fact too narrow to describe the class of systems considered. Automata can model arbitrary algorithms. One of the examples we consider, mutual exclusion, contains very little data but a lot of concurrency and distribution. Thus the term processes is actually more appropriate.

The following subsection introduces a specific model for distributed systems, input/output automata. They were chosen because they cover both the encapsulation and the concurrency aspect and come with a well developed theory of refinement which is very close to the one for applicative data types.

6.1 Input/Output Automata

Input/Output automata were introduced by Lynch and Tuttle [22] for modelling distributed systems. We review only the very basics of the approach. In particular we omit any features dealing with fair computations and restrict ourselves to partial correctness, i.e. safety properties. A complete description of I/O automata can be found in [23].

The interface to an I/O automaton is called an *action signature* which is a set Σ partitioned into *input* actions $in(\Sigma)$, *output* actions $out(\Sigma)$, and *internal* actions $int(\Sigma)$. Output and internal actions are locally controlled, whereas input actions may occur at any point. The union of input and output actions is called *external* actions. A collection of action signatures is called *privacy respecting* if the internal actions of each of them are disjoint from the actions of all others.

An I/O automaton A consists of

- an action signature $sig(A)$,
- a set of states $states(A)$
- a set of start states $start(A) \subseteq states(A)$, and
- a transition relation $s \xrightarrow{\pi}_A s'$, where $s, s' \in states(A)$ and $\pi \in sig(A)$.

In particular we assume that I/O automata are *input-enabled*: for every state s and input action π there is a state s' with $s \xrightarrow{\pi}_A s'$. To simplify some of the definitions, we further assume that if $\pi \notin sig(A)$, $s \xrightarrow{\pi}_A s'$ holds iff $s = s'$. This convention extends to the specifications in later sections were such trivial transitions are left implicit.

If $\gamma = \pi_1 \ldots \pi_n \in sig(A)^*$ and $s_i \xrightarrow{\pi_i}_A s_{i+1}$ we write $s_1 \xrightarrow{\gamma}_A s_{n+1}$. Given a sequence γ and some set S, $\gamma|S$ denotes the restriction of γ to S obtained by deleting all elements from γ which are not in S.

Concurrency is modelled by the composition of automata. A countable collection Σ_i, $i \in I$, of action signatures is called *compatible* if it is privacy respecting and output signatures are pairwise disjoint. Their *composition* $\Sigma = \prod_{i \in I} \Sigma_i$ is defined by

- $in(\Sigma) = \bigcup_{i \in I} in(\Sigma_i) - \bigcup_{i \in I} out(\Sigma_i)$,
- $out(\Sigma) = \bigcup_{i \in I} out(\Sigma_i)$, and
- $int(\Sigma) = \bigcup_{i \in I} int(\Sigma_i)$.

The composition $P = \prod_{i \in I} A_i$ of a countable set of I/O automata A_i, $i \in I$, with compatible action signatures is defined by [6]

- $sig(P) = \prod_{i \in I} \Sigma_i$,
- $states(P) = \prod_{i \in I} states(A_i)$,
- $start(P) = \prod_{i \in I} start(A_i)$, and
- $s \xrightarrow{\pi}_P s'$ iff $s(i) \xrightarrow{\pi}_{A_i} s'(i)$ holds for all $i \in I$.

The distinction between hidden and visible sorts made in Section 5 is unnecessary for I/O automata because all hidden data is concealed in the state. On the other hand, the distinction between internal and external actions serves exactly the same purpose: internal actions are invisible to the environment, i.e. to automata running in parallel.

Implementations of one I/O automaton by another are defined in terms of traces of external, i.e. visible, actions. A sound proof method for implementations are again simulations, called *possibilities mappings* in [22,23]. Given two I/O automata C and A with the same external actions Σ_e, a relation $\sqsubseteq \subseteq states(C) \times states(A)$ is called an *(I/O automaton) simulation* if

- $\forall c \in start(C)\ \exists a \in start(A).\ c \sqsubseteq a$, and
- $\forall \pi \in sig(C).\ a \sqsupseteq c \xrightarrow{\pi}_C c' \Rightarrow \exists \gamma \in sig(A)^*, a'.\ \gamma|\Sigma_e = \pi|\Sigma_e \wedge a \xrightarrow{\gamma}_A a' \sqsupseteq c'$.

Soundness of simulations w.r.t. a trace-based definition of implementation is proved in [22]. However, simulations are not complete, as shown in [25] and Section 7. For a detailed treatment of completeness see Merritt [25].

In the case studies below the situation is somewhat simplified: any state of A is a start state, $\sqsubseteq$ is a total function on reachable states of C, called φ, and A does not have any internal actions. Then φ is a simulation if

$$\forall \pi \in sig(C).\ c \xrightarrow{\pi}_C c' \Rightarrow \varphi(c) \xrightarrow{\pi}_A \varphi(c').$$

This is the usual definition of an automaton homomorphism.

[6] $states(P)$ and $start(P)$ are defined in terms of the ordinary cartesian product.

6.2 Simple Memory

The interface of the simple memory changes considerably when going to a state-based model. The reason is that in an applicative context, passing a parameter to an operation and receiving the return value is an atomic action insofar that no interference is possible. In a state-based model, input and output have to be separated because they constitute independent communications. Hence *read* is split into $read_A(a)$, the environment's request for the datum stored at address a, and $read_D(d)$, the memory's response to that request. Thus $read_D(d)$ fulfills two purposes: it returns the requested datum d and tells the environment that the memory is again "enabled", i.e. further requests can be dealt with. Since writing to memory does not produce any output, an explicit acknowledgement is introduced. Hence there are two further actions: $write(a, d)$, which does the obvious thing, and *written*, which tells the environment that the previous *write* action has been completed successfully. Obviously $read_A$ and *write* are input and $read_D$ and *written* output actions of the memory module.

The introduction of explicit acknowledgement actions is symptomatic of the fact that the memory module is not intended to work properly in any arbitrary environment but only if a certain protocol is followed. Acknowledgement actions are means of establishing the required protocol. In our case the protocol requires that the action trace conforms to the following regular expression:

$$((read_A\ read_D) \mid (write\ written))^* \tag{5}$$

This means the memory can only deal with one read or write request at a time. If the environment issues two consecutive read's without waiting for an answer to the first one, the resulting behaviour is not defined. The following specification makes these informal considerations precise.

Acts = SORTS $(\alpha, \delta)action$
OPS $read_A : \alpha \rightarrow (\alpha, \delta)action$
$read_D : \delta \rightarrow (\alpha, \delta)action$
$write : \alpha * \delta \rightarrow (\alpha, \delta)action$
$written : (\alpha, \delta)action$

SMIO = Map + Acts +
SORTS $(\alpha)cntrl$,
$(\alpha, \delta)state = ((\alpha)cntrl * (\alpha, \delta)map)pair$
OPS $id, ack : (\alpha)cntrl$
$rd : \alpha \rightarrow (\alpha)cntrl$
$<\_, \_, \_> : (\alpha, \delta)state * (\alpha, \delta)action * (\alpha, \delta)state \rightarrow form$
RULES
$<\langle id, m\rangle, read_A(a), \langle s', m'\rangle> \Leftrightarrow s' = rd(a) \wedge m' = m$
$<\langle rd(a), m\rangle, read_D(d), \langle s', m'\rangle> \Leftrightarrow d = m[a] \wedge s' = id \wedge m' = m$
$<\langle id, m\rangle, write(a, d), \langle s', m'\rangle> \Leftrightarrow s' = ack \wedge m' = m[d/a]$
$<\langle ack, m\rangle, written, \langle s', m'\rangle> \Leftrightarrow s' = id \wedge m' = m$

The state of the I/O automaton is a pair of a control state $(\alpha)cntrl$ and a memory $(\alpha, \delta)map$. The control state enforces the right sequence of events, the memory contains the data. The predicate $<\langle s, m\rangle, \pi, \langle s', m'\rangle>$ is a linear form of $\langle s, m\rangle \xrightarrow{\pi} \langle s', m'\rangle$.

Notice that SMIO specifies the transition relation only partially. For example $<\langle rd(a), m\rangle, read_A(b), \langle s', m'\rangle>$ can be neither proved nor disproved. The reason is that issuing a second request $read_A(b)$, while the automaton has not yet answered $read_A(a)$, violates protocol (5). Thus the specification needs to say nothing about it.

Looking at the specification of *write*, one may wonder why *written* is necessary. The effect of *write* is instantaneous and the automaton could have gone back into the idle state immediately, ready for the next request. The inclusion of *written* was a conscious design decision, anticipating that refinements might take rather longer to carry out a *write*. In that case further requests could interrupt a sequence of internal actions completing the *write*. That is in fact what happens in the next refinement step.

6.3 Cache Memory

The I/O automaton implementation of the cache memory is more complex than Lampson's specification. It consists of two separate automata in charge of the cache and the memory respectively. Figure 2 shows their interconnection. The direction of the arrows indicates whether an action should be considered input or output with respect to a particular automaton. The environment communicates only with the cache. If

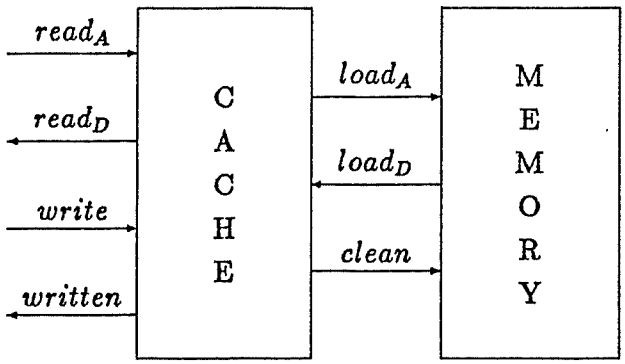

Figure 2: Cache Memory I/O Automaton

possible, all read and write requests are satisfied immediately. Only if an address is not in the cache, does the latter communicate with the memory. Data is loaded into the cache via $load_A/load_D$ and written back via *clean*, all of which are internal actions.

```
CMActs = Acts +
         OPS load_A : α → (α, δ)action
             load_D : δ → (α, δ)action
             clean : α * δ → (α, δ)action
```

In addition we need the following partial specification of sets:

```
Set = FOLE +
      SORTS (α)set
      OPS _ ∈ _ : α * (α)set → form
```

$$\_ + \_, \_ - \_ : (\alpha)set * \alpha \to (\alpha)set$$

RULES

$$x \in (s + y) \Leftrightarrow x \in s \vee x = y$$
$$x \in (s - y) \Leftrightarrow x \in s \wedge x \neq y$$
$$s = t \Leftrightarrow \forall x.\, x \in s \Leftrightarrow x \in t$$

The last law, known as set extensionality, is only used in the section on mutual exclusion.

An now, the cache automaton:

CIO = Map + Set + CMActs +

SORTS $(\alpha, \delta)ccntrl$,

$(\alpha, \delta)cstate = ((\alpha, \delta)ccntrl, (\alpha, \delta)map, (\alpha)set)triple$

OPS $id_c, ack_c : (\alpha, \delta)ccntrl$

$rd_c, ld_c : \alpha \to (\alpha, \delta)ccntrl$

$wr_c : \alpha * \delta \to (\alpha, \delta)ccntrl$

$<\_, \_, \_> : (\alpha, \delta)cstate * (\alpha, \delta)action * (\alpha, \delta)cstate \to form$

RULES

$$<\langle s, c, ds\rangle, read_A(a), C'> \Leftrightarrow s = id_c \wedge C' = \langle rd_c(a), c, ds\rangle$$
$$<\langle s, c, ds\rangle, read_D(d), C'> \Leftrightarrow$$
$$\exists a.\, s = rd_c(a) \wedge D(c, a) \wedge d = c[a] \wedge C' = \langle id_c, c, ds\rangle$$
$$<\langle s, c, ds\rangle, load_A(a), C'> \Leftrightarrow$$
$$s = rd_c(a) \wedge \neg D(c, a) \wedge \exists b.\, D(c, b) \wedge b \notin ds \wedge C' = \langle ld(a), c\backslash b, ds\rangle$$
$$<\langle ld(a), c, ds\rangle, load_D(d), C'> \Leftrightarrow C' = \langle rd_c(a), c[d/a], ds\rangle$$
$$<\langle s, c, ds\rangle, write(a, d), C'> \Leftrightarrow s = id_c \wedge$$
$$\text{if } D(c, a) \to C' = \langle ack_c, c[d/a], ds + a\rangle$$
$$\boxtimes \text{if } \forall b.\, D(c, b) \Rightarrow b \in ds \to C' = \langle wr_c(a, d), c, ds\rangle$$
$$\boxtimes \exists b.\, D(c, b) \wedge b \notin ds \wedge C' = \langle ack_c, (c\backslash b)[d/a], ds + a\rangle \text{ fi fi}$$
$$<\langle s, c, ds\rangle, written, C'> \Leftrightarrow s = ack_c \wedge C' = \langle id_c, c, ds\rangle$$
$$<\langle s, c, ds\rangle, clean(a, d), C'> \Leftrightarrow D(c, a) \wedge d = c[a] \wedge a \in ds \wedge$$
$$((s = id_c \wedge C' = \langle id_c, c, ds - a\rangle) \vee$$
$$((\exists b.\, s = rd_c(b) \wedge \neg D(c, b)) \wedge (\forall b.\, D(c, b) \Rightarrow b \in ds) \wedge$$
$$C' = \langle s, c, ds - a\rangle) \vee$$
$$(\exists b, e.\, s = wr_c(b, e) \wedge C' = \langle ack_c, (c\backslash a)[b/e], ds - a + b\rangle))$$

The state of the cache is a triple: a control state, a cache, and a set of "dirty" addresses. An address is termed dirty if it maps to different data in cache and memory. In that case the cache contains the correct datum which must be written back to memory before the address is overwritten. Keeping track of which cache addresses are dirty reduces communication between memory and cache: clean addresses can be overwritten because they are associated with the right datum in memory.

The transitions under $read_A$, $read_D$ and *written* are straightforward. The $load_A(a)$ action is triggered by $read_A(a)$ in case a is not in the cache. The action $write(a, d)$ can lead to three different responses: if a is in the cache, its contents is overwritten immediately; if it is not in the cache, but there is a clean address b, b is deleted from the cache and (a, d) is added instead. In the worst case, a is not in the cache and all cache addresses are dirty. This prompts the automaton to go into the state $wr_c(a, d)$,

which forces a *clean* action. The action $clean(a, d)$ is triggered by the presence of a dirty address a in the cache. It can occur in three circumstances: spontaneously, if the automaton is currently idle, or when trying to read or write a new address while all cache addresses are dirty.

The alert reader will have spotted that CIO does not specify an I/O automaton because the transition relation is not input-enabled: the specification of $<\langle s, c, ds\rangle, read_A(a), C'>$ says that the input action $read_A$ can only occur if the automaton is in the idle state. Why did we not write $<\langle id_c, c, ds\rangle, read_A(a), C'>$ instead, just as in SMIO, specifying a partial transition relation? Because we would not have arrived at an implementation of SMIO. The reason for this is rather subtle. If $read_A$ occurs when the cache is not in state id_c, it may be thrown into an arbitrary state, including one which does not satisfy the invariant relating dirty set, cache, and memory (see Inv below). But in those states the cache memory stops to behave like a simple memory: it may for example lose data.

This problem is a symptom of the fact that we try to implement a system that is only supposed to work in certain environments, namely those conforming to the protocol (5). A simple solution is to internalize the environment assumptions by adding the environment as a separate I/O automaton to the original specification. Refinement steps would change the memory but not the environment module. One could then show as a lemma that the cache, which is directly driven by the environment, is always in state id_c when a read request arrives.

The memory automaton is much simpler than the cache and bears a strong resemblance to SMIO.

MIO = CMActs +
SORTS $(\alpha)mcntrl$,
$(\alpha, \delta)mstate = ((\alpha)mcntrl, (\alpha, \delta)map)pair$
OPS $id_m : (\alpha)mcntrl$
$ld_m : \alpha \to (\alpha)mcntrl$
$<\_, \_, \_> : (\alpha, \delta)mstate * (\alpha, \delta)action * (\alpha, \delta)mstate \to form$
RULES
$<\langle id_m, m\rangle, load_A(a), m'> \Leftrightarrow m' = \langle ld_m(a), m\rangle$
$<\langle s, m\rangle, load_D(d), m'> \Leftrightarrow \exists a.\ s = ld_m(a) \wedge d = m[a] \wedge m' = \langle id_m, m\rangle$
$<\langle id_m, m\rangle, clean(a, d), m'> \Leftrightarrow m' = \langle id_m, m[a/d]\rangle$

In contrast to MIO there is no acknowledgement action corresponding to *written*. At this stage of the development it is not necessary because *clean* happens instantaneously. Further refinement steps may force the addition of such an acknowledgement action, but for the time being we stay with the simpler model.

The complete cache-memory automaton is the composition of CIO and MIO:

CMIO = CIO + MIO +
SORTS $(\alpha, \delta)cmstate = ((\alpha, \delta)cstate, (\alpha, \delta)mstate)pair$
OPS $\ll\_, \_, \_\gg : (\alpha, \delta)cmstate * (\alpha, \delta)action * (\alpha, \delta)cmstate \to form$
RULES
$\ll\langle c, m\rangle, \pi, \langle c', m'\rangle\gg \Leftrightarrow <c, \pi, c'> \wedge <m, \pi, m'>$

In order to prove correctness some further definitions had to be introduced.

Inv = CMIO +

OPS $inv : (\alpha,\delta)cstate * (\alpha,\delta)mstate \to form$

$con : (\alpha,\delta)map * (\alpha)set * (\alpha,\delta)map \to form$

$f : (\alpha,\delta)ccntrl \to (\alpha)mcntrl$

RULES

$inv(\langle s,c,ds\rangle,\langle t,m\rangle) \Leftrightarrow$

$con(c,ds,m) \wedge (\forall a.\ s = ld_c(a) \Rightarrow \neg D(c,a)) \wedge t = f(s)$

$con(c,ds,m) \Leftrightarrow \forall a.\ D(c,a) \wedge a \notin ds \Rightarrow c[a] = m[a]$

$f(id_c) = f(rd_c(a)) = f(wr_c(a,d)) = f(ack_c) = id_m,\ f(ld_c(a)) = ld_m(a)$

The invariant relates the states of the cache and the memory automata and consists of three parts. The most important one, *con*, characterizes clean addresses: if a cache address is not in the dirty set, it is mapped to the same datum by both cache and memory. The second conjunct asserts that the cache automaton is in the load state only if an address isn't in the cache. The last one, $t = f(s)$, asserts that the memory automaton's control state is a particular function of the cache automaton's control state. Start states of CMIO are identified with those meeting the invariant.

The following lemmas show how one of the invariants is preserved under manipulations of the cache and the dirty set.

$$\begin{aligned}
con(c,ds,m) &\Rightarrow con(c[d/a],ds+a,m)\\
con(c,ds,m) &\Rightarrow con(c\backslash a,ds-a,m)\\
con(c,ds,m) &\Rightarrow con(c\backslash a,ds,m)\\
\neg D(c,a) \wedge con(c,ds,m) &\Rightarrow con(c[m[a]/a],ds,m)\\
D(c,a) \wedge con(c,ds,m) &\Rightarrow con(c,ds-a,m[c[a]/a])\\
D(c,a) \wedge con(c,ds,m) &\Rightarrow con(c\backslash a,ds-a,m[c[a]/a])
\end{aligned}$$

They enable us to prove that inv is in fact invariant:

$$inv(c,m) \wedge \ll\langle c,m\rangle,\pi,\langle c',m'\rangle\gg \Rightarrow inv(c',m')$$

Finally we come to the homomorphisms mapping concrete to abstract states. Because the state of the simple memory automaton is a pair, two separate functions are specified, one for each component.

Hom = SMIO + Inv +

OPS $\varphi_s : (\alpha,\delta)ccntrl \to (\alpha)cntrl$

$\varphi_m : (\alpha,\delta)ccntrl * (\alpha,\delta)map * (\alpha,\delta)map \to (\alpha,\delta)map$

$\varphi : (\alpha,\delta)map * (\alpha,\delta)map \to (\alpha,\delta)map$

RULES

$\varphi_s(id_c)=id,\ \varphi_s(rd_c(a))=\varphi_s(ld_c(a))=rd(a),\ \varphi_s(wr_c(a,d))=\varphi_s(ack_c)=ack$

$\varphi_m(id_c,c,m)=\varphi_m(rd_c(a),c,m)=\varphi_m(ld_c(a),c,m)=\varphi_m(ack_c,c,m)=\varphi(c,m)$

$\varphi_m(wr_c(a,d),c,m) = \varphi(c[d/a],m)$

$\varphi(c,m)[a] = \text{if } D(c,a) \to c[a] \boxtimes m[a] \text{ fi}$

With the help of two further lemmas

$$\varphi(c[d/a],m) = \varphi(c,m)[d/a]$$

$$con(c,ds,m) \wedge D(c,a) \wedge a \notin ds \Rightarrow \varphi(c\backslash a,m) = \varphi(c,m)$$

the main theorem, mapping CMIO to SMIO, can be proved:

$$inv(\langle s, c, ds\rangle, \langle t, m\rangle) \ \wedge \ \ll\langle\langle s, c, ds\rangle, \langle t, m\rangle\rangle, \pi, \langle\langle s', c', ds'\rangle, \langle t', m'\rangle\rangle\gg$$
$$\Rightarrow \ <\langle\varphi_s(s), \varphi_m(s, c, m)\rangle, \pi, \langle\varphi_s(s'), \varphi_m(s', c', m')\rangle>$$

There are two points in the axiomatizations that have been swept under the carpet: equality and exhaustion axioms. More precisely, the specifications also need to state that the actions and states defined are the only ones, and which states and actions are equal or unequal. The former is simply a disjunction, for example $s = id_m \vee \exists a.s = ld_m(a)$ for s of type $(\alpha)mcntrl$. However, stating all possible equalities and inequalities between n constructors requires n^2 axioms, which the reader is spared.

6.4 Mutual Exclusion

A development of coherent caches in the I/O automaton model is considerably more complicated than the applicative version in Section 5.5 because one cannot ignore interference any longer. In fact, it is not even clear in what sense a distributed version of Lampson's original specification implements the simple memory. For those reasons our last example focusses on an archetypical problem in distributed computing: mutual exclusion. Any implementation of the coherent cache memory scheme must contain a solution to this problem because the memory is a centralized resource shared by all processors.

In our formulation of the mutual exclusion problem there are four kinds of actions, indexed by some set ι, the customers to be served: $req(i)$, $do(i)$ and $rel(i)$ indicate the request, the usage and the release of the service by customer i; $grant(i)$ grants the service to customer i.

MutexActs = SORTS $(\iota)action$
OPS $req, grant, do, rel : \iota \rightarrow (\iota)action$

The top level specification defines all legal sequences of these actions. All four actions are output actions of Mutex.

Mutex = Set + MutexActs +
SORTS $(\iota)mcntrl$,
$(\iota)mstate = ((\iota)mcntrl, (\iota)set)pair$
OPS $idle : (\iota)mcntrl$
$active : \iota \rightarrow (\iota)mcntrl$
$<\_,\_,\_> : (\iota)mstate * (\iota)action * (\iota)mstate \rightarrow form$
RULES
$<\langle a, w\rangle, req(i), \langle a', w'\rangle> \ \Leftrightarrow \ i \notin w \wedge a \neq active(i) \wedge a' = a \wedge w' = w + i$
$<\langle a, w\rangle, grant(i), \langle a', w'\rangle> \ \Leftrightarrow \ a = idle \wedge i \in w \wedge a' = active(i) \wedge w' = w - i$
$<\langle a, w\rangle, do(i), \langle a', w'\rangle> \ \Leftrightarrow \ a = active(i) \wedge a' = a \wedge w' = w$
$<\langle a, w\rangle, rel(i), \langle a', w'\rangle> \ \Leftrightarrow \ a = active(i) \wedge a' = idle \wedge w' = w$

The state of the automaton consists of a control state which is either idle or records the currently active customer, and a set of waiting customers. The specification is extremely liberal in that either individual processes may get stuck on the waiting list

because of an unfair selection strategy, or one customer may grab the service, never to release it again.

The distributed implementation of Mutex involves busy waiting. Each customer i is modelled by the finite state I/O automaton in Figure 3. The up and down arrows indicate output and input actions respectively. In addition to the $req(i)$ action, which

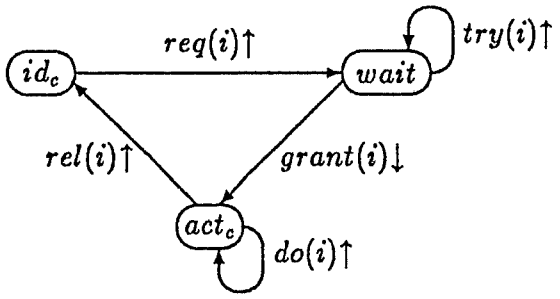

Figure 3: I/O Automaton for Customer i

just announces the intention of grabbing the service, there is a new action $try(i)$ which is repeatedly performed until the service is granted.

MutexActs1 = MutexActs + OPS $try : \iota \rightarrow (\iota)action$

The algebraic specification Customers embodies the composition of all customer automata into one. The state is a map from customers to control states.

Customers = MutexActs1 +
SORTS $ccntrl$
OPS $id_c, wait, act_c : ccntrl$
$<\_,\_,\_> : (\iota, ccntrl)map * action * (\iota, ccntrl)map \rightarrow form$
RULES
$<c, req(i), c'> \Leftrightarrow c[i] = id_c \wedge c' = c[wait/i]$
$<c, try(i), c'> \Leftrightarrow c[i] = wait \wedge c' = c$
$c[i] = wait \Rightarrow <c, grant(i), c'> \Leftrightarrow c' = c[act_c/i]$
$<c, do(i), c'> \Leftrightarrow c[i] = act_c \wedge c' = c$
$<c, rel(i), c'> \Leftrightarrow c[i] = act_c \wedge c' = c[id_c/i]$

The automaton granting access to the service is depicted in Figure 4. The diagram is slightly misleading as there is a state $gr(i)$ for each customer i.

The corresponding algebraic specification is

Service = MutexActs1 +
SORTS $(\iota)scntrl$
OPS $id_s, act_s : (\iota)scntrl$
$gr : \iota \rightarrow (\iota)scntrl$
$<\_,\_,\_> : (\iota)scntrl * action * (\iota)scntrl \rightarrow form$

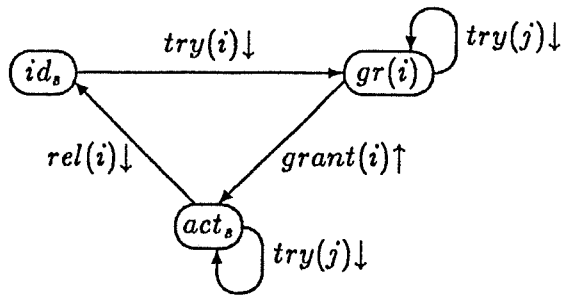

Figure 4: I/O Automaton Granting Access

RULES

$<id_s, try(i), s'> \Leftrightarrow s' = gr(i)$
$<gr(i), try(j), s'> \Leftrightarrow s' = gr(i)$
$<act_s, try(j), s'> \Leftrightarrow s' = act_s$
$<s, grant(i), s'> \Leftrightarrow s = gr(i) \wedge s' = act_s$
$<act_s, rel(i), s'> \Leftrightarrow s' = id_s$
$<s, req(i), s'> \Leftrightarrow s' = s$
$<s, do(i), s'> \Leftrightarrow s' = s$

The complete cache-memory automaton is the composition of Customers and Service:

Mutex1 = Customers + Service +
SORTS $(\iota)state = ((\iota, ccntrl)map, (\iota)scntrl)pair$
OPS $\ll\_,\_,\_\gg : (\iota)state * (\iota)action * (\iota)state \rightarrow form$
RULES
$\ll\langle c, s\rangle, \pi, \langle c', s'\rangle\gg \Leftrightarrow <c, \pi, c'> \wedge <s, \pi, s'>$

The correctness of Mutex1 depends on a number of invariants:

Inv = Mutex1 +
OPS $inv : (\iota, ccntrl)map * (\iota)scntrl \rightarrow form$
$con : (\iota, ccntrl)map \rightarrow form$
RULES
$inv(c, s) \Leftrightarrow con(c) \wedge (\forall i.\ s{=}gr(i) \Rightarrow c[i]{=}wait) \wedge (s \neq act_s \Rightarrow \forall i. c[i] \neq act_c)$
$con(c) \Leftrightarrow \forall i, j.\ c[i] = act_c \wedge c[j] = act_c \Rightarrow i = j$

Consistency (con) says that no two customers can be active at any one time. The other two conjuncts of the invariant assert that if the service is about to be granted to customer i, he must be waiting for it, and that no customer can be active if the service isn't.

With the help of the following simple lemmas,

$$\begin{aligned} con(c) \wedge s \neq act_c &\Rightarrow con(c[s/i]) \\ (\forall i.\ c[i] \neq act_c) &\Rightarrow con(c[act_c/i]) \\ con(c) \wedge c[i] = act_c &\Rightarrow c[j] \neq act_c \Leftrightarrow j \neq i \end{aligned}$$

invariance of inv is readily established:

$$inv(c,s) \wedge \ll\langle c,s\rangle,\pi,\langle c',s'\rangle\gg \;\Rightarrow\; inv(c',s')$$

The need for these invariants becomes apparent during the correctness proof of Mutex1 based on the two homomorphisms φ_c and φ_w which produce the two components of the abstract state, the control state and the waiting set respectively.

Hom = Mutex + Mutex1 +
OPS $\varphi_c : (\iota, ccntrl)map \rightarrow (\iota)mcntrl$
$\varphi_w : (\iota, ccntrl)map \rightarrow (\iota)set$
RULES
$i \in \varphi_w(c) \;\Leftrightarrow\; c[i] = wait$
$con(c) \wedge (\forall i.c[i] \neq act_c) \;\Rightarrow\; \varphi_c(c) = idle$
$con(c) \wedge c[i] = act_c \;\Rightarrow\; \varphi_c(c) = active(i)$

The two lemmas

$$con(c) \;\Rightarrow\; \varphi_c(c) = active(i) \Leftrightarrow c[i] = act_c$$
$$c[i] \neq act_c \wedge s \neq act_c \wedge con(c) \;\Rightarrow\; \varphi_c(c[s/i]) = \varphi_c(c)$$

finally enable us to prove

$$inv(c,s) \wedge \ll\langle c,s\rangle,\pi,\langle c',s'\rangle\gg \;\Rightarrow\; <\langle\varphi_s(c),\varphi_w(c)\rangle,\pi,\langle\varphi_s(c'),\varphi_w(c')\rangle>$$

7 Applicative versus State-Based

Whether a data type should be specified as applicative or state-based is a design decision. It is influenced by the anticipated pattern of usage, the environment of the eventual implementation, resource and reliability considerations. Certain implementation languages may not offer any choice, one way or the other. If there is a choice as in CLU or Standard ML, applicative types can cause excessive copying whereas state-based types bring the dangers of changes by side effect with them. Whatever the eventual choice, it must be part of the specification because it affects whether some implementation is correct or not.

A comparison of applicative and state-based formalisms needs a common framework. We use an imperative language where procedures can have side effects. Data type operations are procedures $p(s,\ldots)$, where s is the data type state that may change as a side effect. The other arguments are of visible type. This set up corresponds to the automaton-based formalism of Section 6 *provided* data type states cannot be copied (duplicated). Otherwise we are in the applicative realm of Section 5: the effect of an operation $p(s,\ldots)$ that returns a new hidden value s' can be modelled by $s' := s; p(s',\ldots)$.

As an example we look at the data type of sets with an operation *pick*. The specification A represents sets by lists; $pick(s)$ returns an arbitrary element of s and removes all its occurrences from s, changing s by side effect. The implementation C is identical to A except that it always *picks* the first element from the list. The only assumption about the other operations is that any permutation of a list that A can generate can be generated by C.

If sets cannot be copied, A and C are behaviourally indistinguishable. If sets can be copied, and s_1 and s_2 are two different copies of the same set, $pick(s_1)$ and $pick(s_2)$ will result in the same elements being picked under interpretation C, but not necessarily so under A. C is still an implementation of A, but not vice versa. A formal demonstration of implementation or non-implementation requires a translation to applicative and automaton-based formalisms which is left to the readers intuition.

In the applicative world, C implements A because the identity relation on lists is a simulation between C and A. A does not implement C because one can easily show that there doesn't exists a simulation in the other direction. In the state-based view, simulations are still a sound implementation criterion (which is why C still implements A) but not a complete one (because there is no simulation between A and C). The issue of completeness is resolved in [15] and [25] by the introduction of the dual of simulation called an "upwards simulation" and a "prophecy mapping" respectively. It can be shown that the relation that pairs two lists iff they contain the same set of elements is an upwards simulation. Thus A also implements C.

A different method of obtaining completeness in the state-based case was studied by Abadi and Lamport [1]. They restrict their simulations to be functions, i.e. homomorphisms, but allow the introduction of auxiliary "history" and "prophecy" variables in the implementation during the correctness proof.

References

[1] M. Abadi, L. Lamport: *The Existence of Refinement Mappings*, Proc. 3rd Symposium Logic in Computer Science (1988), 165-175.

[2] R.S. Boyer, J S. Moore: *A Computational Logic Handbook*, Academic Press (1988).

[3] M. Broy: *A Theory for Nondeterminism, Parallelism, Communication, and Concurrency*, Theoretical Computer Science 45 (1986), 1-61.

[4] M. Broy: *Extensional Behaviour of Concurrent, Nondeterministic, Communicating Systems*, in Control Flow and Data Flow: Concepts of Distributed Programming (M. Broy, ed.), Springer Verlag (1985).

[5] M. Broy, B. Möller, P. Pepper, M. Wirsing: *Algebraic Implementations Preserve Program Correctness*, Science of Computer Programming 7 (1986), 35-53.

[6] R. de Nicola, M.C.B. Hennessy: *Testing Equivalences for Processes*, Proc. 10th ICALP, LNCS 154 (1983), 548-560. Full version in Theoretical Computer Science 34 (1984), 83-133.

[7] E.W. Dijkstra: *A Disciplin of Programming*, Prentice-Hall (1976).

[8] H. Ehrig, B. Mahr: *Fundamentals of Algebraic Specification 1*, EATCS Monograph on Theoretical Computer Science, Springer Verlag (1985).

[9] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, J. Meseguer: *Principles of OBJ2*, Proc. 12th ACM Symposium on Principles of Programming Languages (1985), 52-66.

[10] S.J. Garland, J.V. Guttag: *An Overview of LP, The Larch Prover*, Proc. 3rd Intl. Conf. Rewriting Techniques and Applications, LNCS 355 (1989), 137-151.

[11] Michael J.C. Gordon: *HOL: A Proof Generating System for Higher-Order Logic*, in: Graham Birtwistle and P.A. Subrahmanyam, editors, VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers (1988), 73-128.

[12] G. Hansoul: *Systemes Relationelles Et Algebres Multiformes*, Ph.D. Thesis, Université de Liege, 1979/80.

[13] R. Harper: *Introduction to Standard ML*, Report ECS-LFCS-86-14, Dept. of Comp. Sci., Univ. of Edinburgh, 1986.

[14] C.A.R. Hoare: *Proof of Correctness of Data Representation*, Acta Informatica 1 (1972), 271-281.

[15] J. He, C.A.R. Hoare, J.W. Sanders: *Data Refinement Refined*, Proc. 1st European Symposium on Programming, LNCS 213 (1986).

[16] M.C.B. Hennessy: *Powerdomains and Nondeterministic Recursive Definitions*, Proc. Intl. Symposium on Programming, LNCS 137 (1982), 178-193.

[17] M. Hennessy, R. Milner: *Algebraic Laws for Nondeterminism and Concurrency*, J. ACM Vol. 32, No. 1, January 1985, 137-161.

[18] C.B. Jones: *Systematic Software Development Using VDM*, Prentice-Hall International (1986).

[19] R. Kuiper: *Enforcing Nondeterminism via Linear Temporal Logic Specifications using Hiding*, Proc. Coll. on Temporal Logic and Specification, Altrincham, 1987, to appear in LNCS.

[20] B. Lampson: *Specifying Distributed Systems*, Proc. 1988 Marktoberdorf Summer School, Springer Verlag.

[21] B. Liskov, R. Atkinson, T. Blum, E. Moss, C. Schaffert, R. Scheifler, A. Snyder: *CLU Reference Manual*, LNCS 114 (1981).

[22] N.A. Lynch, M.R. Tuttle: *Hierarchical Correctness Proofs for Distributed Algorithms*, Proc. 6th ACM Symposium on Principles of Distributed Computing, Vancouver, August 1987, 137-151.

[23] N.A. Lynch, M.R. Tuttle: *An Introduction to Input/Output Automata*, Report MIT/LCS/TM-373, Lab. for Computer Science, MIT (1989), to appear in the CWI Quaterly, September 1989.

[24] T.S.E. Maibaum, Pauolo A.S. Veloso, M.R. Sadler: *A Theory of Abstract Data Types for Program Development: Bridging the Gap?*, Proc. TAPSOFT 1985, LNCS 186, 214-230.

[25] M. Merritt: *Completeness Theorems for Automata*, this volume.

[26] G. Nelson: *A Generalization of Dijkstra's Calculus*, Research Report 16, Digital Equipment Corporation, Systems Research Center, April 1987.

[27] T. Nipkow: *Nondeterministic Data Types: Models and Implementations*, Acta Informatica 22 (1986), 629-661.

[28] T. Nipkow: *Are Homomorphisms Sufficient for Behavioural Implementations of Deterministic and Nondeterministic Data Types?*, Proc. 4th Symposium on Theoretical Aspects of Computer Science, LNCS 247 (1987), 260-271.

[29] T. Nipkow: *Behavioural Implementations Concepts for Nondeterministic Data Types*, Ph.D. Thesis, Tech. Rep. UMCS-87-5-3, Dept. of Comp. Sci., The Univ. of Manchester, 1987.

[30] T. Nipkow: *Observing Nondeterministic Data Types*, Proc. 5th Workshop on Specification of Abstract Data Types (1987), LNCS 332, 170-183.

[31] T. Nipkow: *Equational Reasoning in Isabelle*, Science of Computer Programming 12 (1989), 123-149.

[32] T. Nipkow: *Term Rewriting and Beyond – Theorem Proving in Isabelle*, to appear in Formal Aspects of Computer Science.

[33] D.M.R. Park: *Concurrency and Automata on Infinite Sequences*, LNCS 104 (1981).

[34] L.C. Paulson: *Logic and Computation*, Cambridge University Press (1987).

[35] L.C. Paulson: *Isabelle: The next 700 Theorem Provers*, in: P. Odifreddi (editor), *Logic and Computer Science*, Academic Press (1989), in press.

[36] L.C. Paulson: *The Foundation of a Generic Theorem Prover*, Journal of Automated Reasoning (1989), in press.

[37] O. Schoett: *Ein Modulkonzept in der Theorie Abstrakter Datentypen*, Report IfI-HH-B-81/81, Universität Hamburg, Fachbereich Informatik, 1981.

[38] O. Schoett: *Data Abstraction and the Correctness of Modular Programming*, Ph.D. Thesis, Tech. Rep. CST-42-87, Dept. of Comp. Sci., Univ. of Edinburgh, 1987.

[39] D.S. Scott, C.A. Gunter: *Semantic Domains*, to appear in Handbook of Theoretical Computer Science, North-Holland.

[40] R.J. Shoenfield: *Mathematical Logic*, Addison-Wesley (1967).

[41] M.B. Smyth: *Powerdomains*, Journal of Computer and System Science 2 (1978), 23-36.

[42] J.M. Spivey: *The Z Notation: A Reference Manual*, Prentice-Hall International (1989).