Foundations of the Trace Assertion Method of Module Interface Specification [*]

Ryszard Janicki
McMaster University
Hamilton, Ontario, Canada L8S 4K1
janicki@mcmaster.ca

Emil Sekerinski
McMaster University
Hamilton, Ontario, Canada L8S 4K1
emil@mcmaster.ca

June 2000

Abstract

The trace assertion method is a formal state machine based method for specifying module interfaces. A module interface specification treats the module as a black-box, identifying all module's access programs (i.e. programs that can be invoked from outside of the module), and describing their externally visible effects. In the method, both the module states and the behaviors observed are fully described by traces built from access program invocations and their visible effects. A formal model for the trace assertion method is proposed. The concept of step-traces is introduced and applied. The stepwise refinement of trace assertion specifications is considered. The role of non-determinism, normal and exceptional behavior, value functions and multi-object modules are discussed. The relationship with algebraic specifications is analyzed. A tabular notation for writing trace specifications to ensure readability is adapted.

Keywords. Module interface specifications, trace assertion method, state machines, Mealy machines, step-sequences, relational model, nondeterminism, module refinement, tabular notation.

1 Introduction

Software modules, viewed as "black boxes" [25, 23], hide some design decisions and provide abstract data types. They can be specified using the *trace assertion method*. A trace is a complete history of the visible behavior of a module. It includes all events affecting the module, eventually with the outputs produced. Formally a trace is a sequence of events. The fundamental principle is that a trace specification describes only those features of a module (or an object in general) that are externally observable and the central idea of the approach is that traces can be divided into clusters and each cluster is represented by a single canonical trace.

The trace assertion method was first formulated by Bartussek and Parnas in [3], as a possible answer for problems with algebraic specifications [7, 34], which will be discussed later. It also can avoid the problem of overspecification in model-oriented specifications, e.g. [1]. A typical example is the use of a sequence for specifying a stack module, where *PUSH* will append the new element either at the front or the tail of the sequence, the choice being arbitrary. In the trace assertion method, this decision

[*]To appear in IEEE Transactions on Software Engineering. Partially supported by NSERC Research Grant

is avoided. Since its introduction the method has undergone many modifications [12, 22, 27, 33]. In recent years, there has been an increased interest in the trace assertion method [14, 16, 15, 24, 28, 32]. However, a satisfactory foundation has not yet been developed.

The trace assertion method is based on the following postulates:

- *Information hiding* [25, 23] is a fundamental principle for specification.
- *Sequences* are simple and powerful tool for specifying abstract *objects*.
- *Explicit equations* are preferable over *implicit equations* like those of algebraic specifications.
- *State machines* are simple and powerful tool for specifying *modules*.

For many applications state machines are better than algebras, and their use for specification is growing [1, 2, 11]. State machines (not necessary finite) are equivalent to algebras. This relationship differs for different machines and algebras, but the general idea of relationship may be illustrated as follows:

$$\underbrace{\delta(p,a) = q}_{\textit{state machine}} \quad \Leftrightarrow \quad \underbrace{a(p) = q}_{\textit{algebra}},$$

where δ is a transition function of a state machine with a as a function name, and $a(p)$ is a function named a applied to p. See [8, 4] and Section 15.

The term "trace" has at least two different meanings. One it that a trace is just a sequence of events, actions, operations, or systems calls, i.e. it is a sequence of specially interpreted elements. The other meaning is that a trace is an element of a partially commutative monoid, where the monoid operation is concatenation (see [6]). In the second case the name "Mazurkiewicz traces" is often used [6, 18] . Traces in the first sense can be treated as a special case of the second (the independency relation is empty, i.e. no commutativity at all). The "step-traces" used in this paper lie somewhere between the first and the second meaning.

The contributions of this paper to the trace assertion method are:

- The role of *nondeterminism*, which caused some problem in the previous models is explained.
- The concept of *exceptional behavior* if formally analyzed.
- The role of value functions, in particular for nondeterministic modules, is discussed.
- The use of *step-traces* to overcome difficulties with asymmetry caused by the use of ordinary traces is proposed.
- A notion of *refinement* for trace assertion specifications is introduced.
- The use of *abstract state constructors* for the problem of finding canonical traces is certain situations is suggested.
- A formal model for *multi-object* modules is proposed and discussed.

Overview. In the next section we introduce and briefly discuss three simple modules. These modules are used to illustrate the major problems and solutions. In Section 3 the question "What is an atomic observable event?" is discussed. Section 4 reviews the fundamentals of the relational model of programs and of program refinement. The automata model for module specifications is introduced in Section 5, together with a notion of module refinement and a simulation condition. A module access-program may return some values, but is it absolutely necessary to specify this fact by a separate output value function? This problem is discussed in Section 6. Objects described by sequences and the concept of step-sequences are discussed in Section 7, while automata with states specified by the step-sequences are analyzed in Section 8. The formal concept of a trace assertion specification is given in Section 9. The special instances of the trace assertion specification in Mealy form and the controversial use of invisible actions are discussed respectively in Sections 10 and 11. Exceptional behavior is discussed in Section 12. The idea is that misuses can be modeled separately and eventually they may be added to the pure trace specification as an enhancement. Section 13 defines a format for the trace specification technique. All the examples from Section 2 are formally specified in this format. Refinement with these examples is illustrated in Section 14. Section 15 deals with multi-object modules. The uniquely labeled sets of step-traces are introduced and used as a specification tool. The relationship between trace assertions and algebraic specifications is analyzed in Section 16. The last section contains final comments.

2 Introductory Examples

We shall use the following examples of modules: Stack, Unique Integer, Very Drunk Stack and Drunk Stack. Each module is designed to implement a single object. The Stack module provides three access programs,

- $\mathit{PUSH}(i)$: enters an integer *i* on the stack,
- *POP*: takes no arguments and removes the top of the stack, and
- *TOP*: takes no arguments and returns the value which is on the top of the stack.

Intuitively, a state of the stack is determined by the finite sequence of integers, the last element of the sequence represents the top of the stack, and the first represents the bottom. Note that every sequence of properly used access programs leads to exactly one state. For instance $\mathit{PUSH}(4).\mathit{PUSH}(1).\mathit{POP}.\mathit{PUSH}(7).\mathit{TOP}$ and $\mathit{PUSH}(4).\mathit{PUSH}(7)$ both lead to the state $\langle 4,7\rangle$. They could be seen as equivalent and we can choose for instance the trace $\mathit{PUSH}(4).\mathit{PUSH}(7)$ as a *canonical trace* representing the state $\langle 4,7\rangle$.

The Unique Integer module provides only one access program,

- *GET*: does not take any argument and returns an integer value from the set of integers a machine can represent.

The only restriction on the return value is that it cannot be any value that was returned by previous *GET* invocations. Intuitively the state of Unique Integer is determined by the set of all integers that were returned by all previous *GET* invocations. In this case the sequence, say $\mathit{GET}.\mathit{GET}.\mathit{GET}$, corresponds to any set $\{i_1, i_2, i_3\}$, where i_1, i_2 and i_3 are distinct integers. However, the invocation of *GET* is only a part of a single observable event, an invocation of *GET* returns an integer *i*, so the full observable event is a pair (GET, i), or, more conveniently, *GET*:*i*. A pair *GET*:*i* is a call-response event, with the call *GET* and the response *i*. Any trace built from *GET*:*i* pairs describes one state. For instance both *GET*:5.*GET*:1.*GET*:8, and *GET*:1.*GET*:5.*GET*:8, describe the state $\{1, 5, 8\}$. They could be seen as equivalent and we can choose for instance *GET*:1.*GET*:5.*GET*:8 as the canonical trace. However, since the order of *GET*'s is not important, quite opposite, it may cause some problems when imposed, we will use a canonical *step-trace* $\langle GET:1.GET:5.GET:8\rangle$, as a state descriptor. The operator $\langle\cdot\rangle$ makes the order irrelevant, i.e. $\langle GET:1.GET:5.GET:8\rangle = \langle GET:8.GET:1.GET:5\rangle$, etc. (see Section 7).

Both Stack and Unique Integer can be modeled by state machines (automata) for which every trace describes exactly one state. The difference is that for Unique Integer traces are built from pairs $(call, response)$ while for Stack calls alone are sufficient. The case when traces built from calls alone are sufficient are called *output independent*.

Module Drunk Stack is the same as Stack except that access program *POP* behaves differently,

- *POP*: if the length of the stack is one removes the top element, if the length is greater than one removes either the top element or the two top elements of the stack.

Now the trace $PUSH(7).PUSH(4).PUSH(1).PUSH(3).POP$ may lead to two states: $\langle 7, 4, 1\rangle$ and $\langle 7, 4\rangle$. Adding outputs to the events does not change the situation since both *PUSH* and *POP* produce no output. However, each state is unambiguously described by an appropriate trace built from *PUSH* calls. For instance $PUSH(7).PUSH(4).PUSH(1)$ describes the state $\langle 7, 4, 1\rangle$, and only this state, so canonical traces can be built. However, the traces $PUSH(7).PUSH(4).PUSH(1)$ and $PUSH(7).PUSH(4).PUSH(1).$ $PUSH(3).POP$ may no longer be considered as equivalent, they lead to different sets of states. They could be interpreted as *similar* since the sets of states they represent are not disjoint, and they both belong to the same *cluster* of traces. The cluster of traces they belong to is the set of all traces that may lead to the state $\langle 7, 4, 1\rangle$. This cluster is unambiguously represented by the trace $PUSH(7).PUSH(4).PUSH(1)$. The program *PUSH* is "sober" so it can be used to specify canonical traces.

The use of output independent traces is sufficient for Drunk Stack. It can also be modeled by a non-deterministic state machine with states unambiguously described by canonical traces.

The Very Drunk Stack has two "drunk" access programs *POP* and *PUSH*. Access programs *TOP* and *POP* are the same as for Drunk Stack, while the behavior of *PUSH* is the following,

- $PUSH(i)$: enters an integer *i* either once or twice on the stack.

In this case the trace $PUSH(7).PUSH(4)$ leads to $\langle 7, 4\rangle$, $\langle 7, 7, 4\rangle$, $\langle 7, 4, 4\rangle$, or $\langle 7, 7, 4, 4\rangle$. Moreover, each trace which does not lead to the empty stack, may lead to at least two different states. Thus canonical traces, interpreted as traces that can unambiguously describe states, cannot be defined. We need to proceed differently. One way is to observe that the state $\langle 7, 4\rangle$ is the only state that can be reached by both the trace $PUSH(7).PUSH(4)$ and the trace $PUSH(7).PUSH(4).POP.POP$. Thus the set of traces

$$\{PUSH(7).PUSH(4)\ ,\ PUSH(7).PUSH(4).POP.POP\}$$

could be used as a trace descriptor of the state $\langle 7, 4\rangle$. One may observe that every state can unambiguously described in this sense by a finite set of traces. Modeling states of modules by sets of canonical traces was proposed in [24]. However, we reject such an approach. The sets of traces that describe states can be large and complex even for relatively simple, non-deterministic modules. We believe such an approach will result in a complex and unreadable specification. We propose the use of abstract constructor programs instead. In the case of Very Drunk Stack, all states can easily be specified by an *abstract constructor* (invisible) program $push1(i)$ which pushes i exactly once on the stack. The specification obtained is simple and natural (see Section 13, Figures 8 and 9).

3 Alphabet

Since a trace specification describes only those features of a module that are externally observable, the question arises what an atomic observation is. What constitutes an alphabet from which the traces are built? We consider two kinds of observations:

- call events like $PUSH(5)$, and
- call-response events like $GET{:}5$.

Let f be the name of an access program and let $input(f)$ and $output(f)$ be the sets of possible argument and result values. The *signature* $sig(f)$ is the triple:

$$sig(f) = (f, input(f), output(f)).$$

We assume that neither $input(f)$ nor $output(f)$ are empty by having $input(f) = \{nil\}$ and $output(f) = \{nil\}$ as default. For example:

$$\begin{aligned} sig(PUSH) &= (PUSH, integer, \{nil\}), \\ sig(TOP) &= (TOP, \{nil\}, integer), \\ sig(POP) &= (POP, \{nil\}, \{nil\}). \end{aligned}$$

For a finite set E of access program names, the *signature* $sig(E)$ is the set of all signatures of $f \in E$:

$$sig(E) = \{sig(f) \mid f \in E\}.$$

Given E, the *call-response alphabet* Δ_E is the set of all possible triples, written $f(x){:}g$ of access program names, arguments, and return values:

$$\Delta_E = \{f(x){:}g \mid f \in E, x \in input(f), y \in output(f)\}.$$

We adopt the convention of omitting *nil* in signatures. For example, for the stack modules we have $E = \{PUSH, TOP, POP\}$ and:

$$\Delta_E = \{PUSH(i) \mid i \in integer\} \cup \{TOP{:}i \mid i \in integer\} \cup \{POP\}.$$

For a given set E of access program names, we also define the *call alphabet* Σ_E and the *response alphabet* $\mathcal{O}_E$:

$$\begin{aligned} \Sigma_E &= \{f(x) \mid f \in E, x \in input(f)\}, \\ \mathcal{O}_E &= \{d \mid \exists f \in E \,.\, d \in output(f)\}. \end{aligned}$$

Note that the sequences of call-response event occurrences are what is really observed. However, one may abstract away from the output values, if states can be unambiguously described by sequences of call event occurrences only.

4 Relational Model of Programs

We review the fundamentals of the relational model of programs (e.g. [30]). Data refinement is introduced according to [10], except that, rather than taking relations extended by a bottom element, “demonic relational composition” and “demonic refinement” is used.

We write $S \leftrightarrow T$ for the set of all relations between S and T, formally defined as $S \leftrightarrow T = 2^{S\times T}$. For relations $Q \in S \leftrightarrow T$ and $R \in T \leftrightarrow U$, the relational composition $Q \circ R$, the relational image $Q[s]$ of a set $s \subseteq S$, and the relational image $Q(x)$ of an element $x \in S$ are defined as follows:

$$\begin{aligned} Q\circ R &= \{(x,z) \mid \exists y \,.\, x\,Q\,y \wedge y\,R\,z\}, \\ Q[s] &= \{y \mid \exists x \,.\, x\,Q\,y \wedge x \in s\}, \\ Q(x) &= \{y \mid x\,Q\,y\}. \end{aligned}$$

Here, $x\ Q\ y$ stands for $(x,y) \in Q$. If Q is interpreted as a (possibly nondeterministic) program over initial state space S and final state space T, then the *domain* of Q, i.e. the set of all initial states which are related to at least one final state, is the precondition for which execution of Q will terminate with a defined outcome. Outside its domain, program Q may not terminate.

Using the notation $Q(x)$ for the image of x under R suggests that we may equivalently view Q as a set valued function. In particular, where convenient, we define a relation Q by an equation of the from $Q(x) = e$ for all x.

The sequential (demonic) composition $Q\,;R$ is $Q \circ R$ restricted to those initial states for which Q leads to intermediate states in which R is defined. If $x\ Q\ y$ and y is not in the domain of R, then x is not in the domain of $Q\,;R$:

$$Q\,;R = \{(x,z) \mid x(Q\circ R)z \wedge (\forall y \,.\, x\,Q\,y \Rightarrow R(y) \neq \emptyset)\}.$$

Assume $Q, Q' \in S \leftrightarrow S$. Relation Q' is an (algorithmic) refinement of Q if Q' is “more deterministic” than Q and the domain of Q' is not smaller than the domain of Q:

$$Q \sqsubseteq Q' \Leftrightarrow (\forall x \,.\, Q(x) \neq \emptyset \Rightarrow Q'(x) \subseteq Q(x) \wedge Q'(x) \neq \emptyset).$$

Now assume that Q is as above and $Q' \in S' \leftrightarrow S'$. Let R be a relation between the state spaces of Q and Q', i.e. $R \in S \leftrightarrow S'$. Then Q' is a data refinement of Q via R means:

$$Q \sqsubseteq_R Q' \Leftrightarrow (\forall x \,.\, Q(x) \neq \emptyset \Rightarrow (R\circ Q')(x) \subseteq (Q\circ R)(x) \wedge (R\circ Q')(x) \neq \emptyset).$$

Algorithmic refinement is a special case of data refinement, $Q \sqsubseteq_{Id} Q' \Leftrightarrow Q \sqsubseteq Q'$ where Id is the identity relation. Refinement is reflexive and transitive in the sense that $Q \sqsubseteq Q$ and for $Q'' \in S'' \leftrightarrow S''$ and $R' \in S' \leftrightarrow S''$:

$$Q \sqsubseteq_R Q' \wedge Q' \sqsubseteq_{R'} Q'' \Rightarrow Q \sqsubseteq_{R\circ R'} Q''.$$

Sequential composition is monotonic with respect to refinement:

$$P \sqsubseteq_R P' \wedge Q \sqsubseteq_R Q' \Rightarrow P\,;Q \sqsubseteq_R P'\,;Q'.$$

5 Automata

The standard automata model is used for module specifications by associating signatures with the alphabet, similarly to [20]. Data refinement is used for forward simulation of automata. Simulations of automata are further discussed in in [10, 21].

Let Δ be an alphabet, Δ^* be the set of all sequences built from the elements of Δ including the empty sequence denoted by ε. For every two sequences $x, y \in \Delta^*$, their concatenation is denoted by $x.y$. A (nondeterministic) automaton A is a quadruple,

$$A = (\Delta, S, \varrho, s_0),$$

where Δ is the alphabet, S is the (finite or infinite) set of states, ϱ is the transition relation, $\varrho \in \Delta \to S \leftrightarrow S$, and $s_0 \in S$ is the initial state.

The extended transition relation $\varrho^* \in \Delta^* \to S \leftrightarrow S$, is defined for every $x \in \Delta^*$ and $a \in \Delta$ as:

$$\begin{aligned} \varrho^*(\varepsilon) &= Id, \\ \varrho^*(x.a) &= \varrho^*(x) \; ; \varrho(a). \end{aligned}$$

We use automata for specifying modules: The set Δ consists of sequences of call-responses, the set S is the state private to the module in the sense that it is only accessed through calls to the module, the state s_0 is the initial state of the module, and the function ϱ specifies the change of the module's state for each possible call. Formally, for a given signature E, a module specification A is an automaton:

$$A = (\Delta_E, S, \varrho, s_0).$$

The set $L(A) = \{x \in \Delta^* \mid \varrho^*(x)(s_0) \neq \emptyset\}$ contains all valid sequences of call-responses of the module, i.e. describes the *normal behavior* of the module.

Module A is *transition deterministic* if $|\varrho(b)(s)| \leq 1$ for all $b \in \Delta_E$ and $s \in S$. Module A is *output deterministic* if for all $a \in \Sigma_E$ there exists at most one $d \in \mathcal{O}_E$ such that $\varrho(a{:}d) \neq \emptyset$, where $\emptyset$ is the empty relation. Module A is *deterministic* if it is both transition deterministic and output deterministic.

Abstraction in the module specification is achieved in two ways. First, the automaton may be nondeterministic, thus hiding implementation decisions. Secondly, the automaton can use a more abstract state space than would be required for an (efficient) implementation. Abstraction is formalized by introducing a refinement relation between modules. Module refinement is defined in terms of the observable behavior, which ultimately are the possible values returned after a sequence of calls. We first decompose ϱ into a transition relation $\delta_A \in \Sigma \to (S \leftrightarrow S)$, or δ for short, and a value relation $v_A \in \Sigma \to (S \leftrightarrow \mathcal{O})$, or v for short:

$$\begin{aligned} \delta(a) &= \bigcup\{\varrho(a{:}d) \mid d \in \mathcal{O}\}, \\ v(a) &= \{(s, d) \mid \varrho(a{:}d)(s) \neq \emptyset\}. \end{aligned}$$

The extended transition relation $\delta^* \in \Sigma^* \to (S \leftrightarrow S)$, is defined by $\delta^*(\varepsilon) = Id$ and $\delta^*(x.a) = \delta^*(x) \; ; \delta(a)$. The response relation $r_A \in \Sigma^+ \leftrightarrow \mathcal{O}$, or r for short, defines the set of all possible responses (outputs) after a non-empty sequence of calls, starting from the initial state:

$$r(x.a) = (\delta^*(x) \; ; v(a))(s_0).$$

For a given signature E, let $A = (\Delta_E, S, \varrho, s_0)$ and $A' = (\Delta_E, S', \varrho', s'_0)$ be module specifications with the same alphabet but possibly different state space, transition function and initial state. Then A' is a *behavioral refinement* of A, written $A \leq A'$, if after any sequence of calls for which A returns some defined output, A' returns also at least one output value and all the possible outputs returned by A' would also possible for A:

$$A \leq A' \Leftrightarrow r_A \sqsubseteq r_{A'}.$$

Note that behavioral refinement is expressed without direct reference to the states of A and A'. If follows immediately that behavioral refinement is reflexive and transitive. For example, given appropriate definitions of the modules, we would have:

$$\text{Very Drunk Stack} \leq \text{Drunk Stack} \quad \text{and} \quad \text{Drunk Stack} \leq \text{Stack}.$$

Let E be a signature, let $\varrho \in \Delta_E \to S \leftrightarrow S$ and $\varrho' \in \Delta_E \to S' \leftrightarrow S'$ be transition relations with the same alphabet but different state space, and let R be a relation between S and S'. We note that $\Delta_E \subseteq \Sigma_E \times \mathcal{O}_E$. Transition relation ϱ is data-refined by ϱ', written $\varrho \sqsubseteq_R \varrho'$, means that for a given initial state and access program call, the outputs which are possible with ϱ' are also possible with ϱ (the nondeterminism in selecting a response may be reduced) and the final states which are possible for ϱ' are also possible with ϱ (the nondeterminism in selecting a final state may be reduced), where the initial and final states are related via R. Moreover, whenever for a given initial state and access program call at least one response and final state are defined in ϱ, there must be also at least one response and final state defined by ϱ' (the domain must not be reduced), where the initial and final stated are related via R. For this, let $\tilde{\varrho} \in \Sigma_E \times S \leftrightarrow \mathcal{O}_E \times S$ be a relation which is isomorphic to ϱ but makes Σ_E part of the initial state space and $\mathcal{O}_E$ part of the final state space. Data refinement is defined in terms of $\tilde{\varrho}$ and $\tilde{\varrho'}$:

$$\begin{array}{ll} \varrho \sqsubseteq_R \varrho' & \Leftrightarrow \tilde{\varrho} \sqsubseteq_{Id \times R} \tilde{\varrho'}, \\ (a,s)\ \tilde{\varrho}\ (d,t) & \Leftrightarrow s\ (\varrho(a{:}d))\ t. \end{array}$$

Module specification A' *simulates* A via simulation relation R if the initial values are in relation R and the transition relations are data refined via R:

$$A \sqsubseteq_R A' \Leftrightarrow s_0\ R\ s'_0 \wedge \varrho \sqsubseteq_R \varrho'.$$

If for some relation R module A' simulates module A, then A' is a behavioral refinement of A. Hence this gives a practical way of establishing module refinement:

Proposition 5.1 *For a given signature E, let $A = (\Delta_E, S, \varrho, s_0)$ and $A' = (\Delta_E, S', \varrho', s'_0)$ be module specifications. If $R \in S \leftrightarrow S'$ then:*

$$A \sqsubseteq_R A' \Rightarrow A \leq A'.$$

For the purpose of the proof we need two lemmas. First, we generalize data refinement to allow different relations for the initial and final state space. Assume $Q \in S_0 \leftrightarrow S_1$, $Q' \in S'_0 \leftrightarrow S'_1$, $T \in S_0 \leftrightarrow S'_0$, and $U \in S_1 \leftrightarrow S'_1$. Data refinement of Q by Q' via T, U is defined by:

$$Q \sqsubseteq_{T,U} Q' \Leftrightarrow (\forall x \,.\, Q(x) \neq \emptyset \Rightarrow (T \circ Q')(x) \subseteq (Q \circ U)(x) \wedge (T \circ Q')(x) \neq \emptyset).$$

Ordinary data refinement is a special case since $Q \sqsubseteq_T Q' \Leftrightarrow Q \sqsubseteq_{T,T} Q'$. Sequential composition is monotonic with respect to generalized data refinement in the sense that, assuming additionally $R \in S_1 \leftrightarrow S_2$, $R' \in S'_1 \leftrightarrow S'_2$, and $V \in S_2 \leftrightarrow S'_2$:

$$Q \sqsubseteq_{T,U} Q' \wedge R \sqsubseteq_{U,V} R' \Rightarrow Q \,;\, R \sqsubseteq_{T,V} Q' \,;\, R'.$$

For removing a data refinement on the initial state we have that for any Q, R, x, x':

$$x \mathrel{R} x' \wedge Q \sqsubseteq_{R,Id} Q' \Rightarrow (Q(x) \neq \emptyset \Rightarrow Q'(x') \subseteq Q(x) \wedge Q'(x') \neq \emptyset).$$

Proof of Proposition 5.1 First, we observe that by definition, $A \sqsubseteq_R A'$ implies $\tilde{\varrho} \sqsubseteq_{Id \times R} \tilde{\varrho}'$. From this, we can show that for any $a \in \Sigma$:

$$\bigcup\{\varrho(a{:}d) \mid d \in \mathcal{O}\} \sqsubseteq_R \bigcup\{\varrho'(a{:}d) \mid d \in \mathcal{O}\}.$$

This is done by expanding the definitions and some subsequent simplifications. By the definition of $\delta_A(a)$, this is equivalent to:

$$\forall a \in \Sigma \ . \ \delta_A(a) \sqsubseteq_R \delta_{A'}(a).$$

With the monotonicity of sequential composition and by using induction over the length of $x \in \Sigma^*$ we conclude:

$$\forall x \in \Sigma^* \ . \ \delta_A^*(x) \sqsubseteq_R \delta_{A'}^*(x).$$

Furthermore, from the definition of v_A and $\tilde{\varrho} \sqsubseteq_{Id \times R} \tilde{\varrho}'$ we get:

$$\forall a \in \Sigma \ . \ v_A(a) \sqsubseteq_{R,Id} v_{A'}(a).$$

Again, this is done by expanding the definitions and some subsequent simplifications. From these two, using the monotonicity lemma above, we have:

$$\forall x \in \Sigma^*, a \in \Sigma \ . \ \delta_A^*(x) \ ; v_A(a) \sqsubseteq_{R,Id} \delta_{A'}^*(x) \ ; v_{A'}(a).$$

As $s_0 \ R \ s_0'$ holds by the assumption that $A \sqsubseteq_R A'$, we can apply above lemma for removing data refinement on the initial state and get:

$$\begin{aligned} &\forall x \in \Sigma^*, a \in \Sigma \ . \ (\delta_A^*(x) \ ; v_A(a))(s_0) \neq \emptyset \Rightarrow \\ &\quad (\delta_{A'}^*(x) \ ; v_{A'}(a))(s_0') \subseteq (\delta_A^*(x) \ ; v_A(a))(s_0) \wedge (\delta_{A'}^*(x) \ ; v_{A'}(a))(s_0') \neq \emptyset. \end{aligned}$$

By the definition of r_A this is equivalent to:

$$\forall x \in \Sigma^*, a \in \Sigma \ . \ r_A(x.a) \neq \emptyset \Rightarrow r_{A'}(x.a) \subseteq r_A(x.a) \wedge r_{A'}(x.a) \neq \emptyset,$$

which again is equivalent to $r_A \sqsubseteq r_{A'}$, and hence implies $A \leq A'$. ■

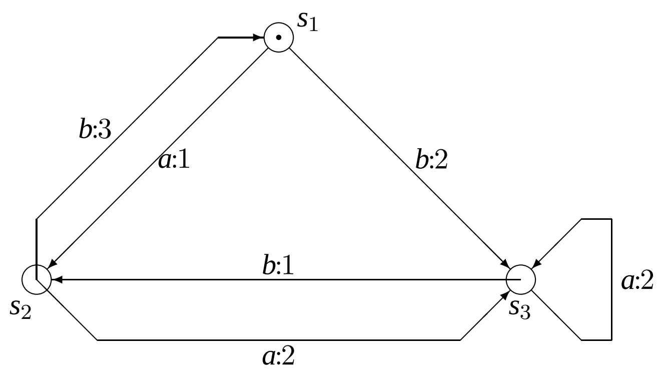
6 Mealy Machines

In contrast to automata, Mealy machines specify the next state and the output by separate functions. For a given signature E, a Mealy machine M is a tuple,

$$M = (\Delta_E, S, \delta, v, s_0)$$

where Δ_E are call-responses of signature E, S is the (finite or infinite) set of states, $\delta \in \Sigma_E \to S \leftrightarrow S$ is the state transition relation, $v \in \Sigma_E \to S \to \mathcal{O}_E$, $s_0 \in S$ is the initial state, and all valid call-responses of δ and v are according to Δ_E:

$$\forall a \in \Sigma_E, s \in S \ . \ \delta(a)(s) \neq \emptyset \Rightarrow a{:}v(a)(s) \in \Delta_E$$



$$\left.\begin{array}{r} \delta(b)(s_1) = \{s_3\} \\ v(b)(s_1) = 2 \end{array}\right\} \quad \Leftrightarrow \quad s_1 \xrightarrow{b:2} s_3 \quad \Leftrightarrow \quad \varrho(b{:}2)(s_1) = \{s_3\}$$

Figure 1: The Mealy machine with $\Sigma = \{a, b\}$, $\mathcal{O} = \{1, 2, 3\}$, or the *deterministic* automaton with $\Delta = \{a{:}1, a{:}2, b{:}1, b{:}2, b{:}3\}$. The state s_1 is initial.

For every Mealy machine M we can construct an automaton A_M,

$$A_M = (\Delta_E, S, \varrho, s_0)$$

over the same alphabet Δ_E, same set S of states, same initial state s_0 and the transition relation $\varrho \in \Delta_E \to S \leftrightarrow S$ defined by:

$$\varrho(a{:}d)(s) = \begin{cases} \delta(a)(s) & \text{if } v(a)(s) = d \\ \emptyset & \text{if } v(a)(s) \neq d \end{cases}$$

The automaton A_M is equivalent to the Mealy machine M in the sense that the set of valid call-response sequences of M and A_M are identical. Figure 1 illustrates the relationship between M and A_M. However, not every automaton, even not every deterministic automaton, can be interpreted as a Mealy machine. In Figure 1, if one adds an arrow from s_1 to s_3 labeled by a:2, the new automaton cannot be interpreted as a Mealy machine [1].

We may use both Mealy machines and standard automata as the backbone of our model. The descriptive power of Mealy machines is at best the same as transition deterministic automata, only notation is different, more complex in our opinion. It might occasionally be convenient to use Mealy machines instead of standard automata. As an example, we define simulation of Mealy machines which implies simulation of the corresponding automata. Assuming $M = (\Delta_E, S, \delta, v, s_0)$ and $M' = (\Delta_E, S', \delta', v', s'_0)$ we define:

$$\begin{array}{ll} M \sqsubseteq_R M' & \Leftrightarrow s_0 \; R \; s'_0 \wedge \delta \sqsubseteq_R \delta' \wedge v \sqsubseteq_R v' \\ v \sqsubseteq_R v' & \Leftrightarrow \forall s \in S, s' \in S', a \in \Sigma_E \; . \; s \; R \; s' \Rightarrow v(a)(s) = v'(a)(s') \end{array}$$

Proposition 6.1 $M \sqsubseteq_R M' \Rightarrow A_M \sqsubseteq_R A_{M'}$.

[1]Since $v \in \Sigma_E \to S \to \mathcal{O}_E$, then $v(s, a)$ can be equal to 1 or 2 but not both. Extending v to $\Sigma_E \to (S \leftrightarrow \mathcal{O})$ does not help, since it does not indicate that a:1 leads from s_1 to s_2 and a:2 from s_1 to s_3.

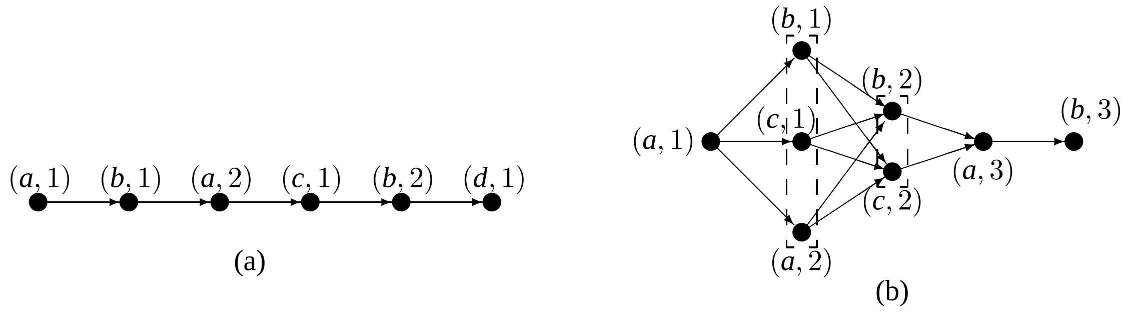


Figure 2: (a) Total order defined by the sequence $a.b.a.c.b.d$ and (b) Weak order defined by the sequence $a.\langle a.c.b\rangle.\langle b.c\rangle.a.b$.

Proof Suppose $M = (\Delta_E, S, \delta, v, s_0)$ and $M' = (\Delta_E, S', \delta', v, s_0')$. We have to show that $s_0\ R\ s_0'$ and $\varrho \sqsubseteq_R \varrho'$ where $\varrho(a{:}d)(s) = \delta(a)(s)$ if $v(a)(s) = d$ and $\emptyset$ otherwise, and similarly $\varrho'(a{:}d)(s) = \delta'(a)(s)$ if $v'(a)(s) = d$ and $\emptyset$ otherwise. The first part follows immediately from $M \sqsubseteq_R M'$, the second part can be shown to hold by first unfolding the definitions. ■

The difference is that here the refinements of δ and v are dealt with separately, which may be of practical advantage.

In general the standard automata provide a better and simpler model. In particular adding non-determinism to value functions in Mealy formalism is problematic and, although possible, is seldom done, because the formalism becomes complex. In [27, 33, 15, 24] Mealy machines were used and we believed that resulted in unnecessary complexity and formal problems [33].

After deciding to use automata as a backbone of the specification technique, the next question is how to describe the set of states in an as abstract as possible way, i.e. in a way which does not commit to implementation decisions prematurely.

7 Defining Objects by Sequences

The ingenuity of the trace assertion method ([3]) is to use traces (i.e. some kind of sequences) not only as a medium to describe behavior, but to specify states as well. Sequences are easy to specify and understand and, since we observe only traces of call-responses, they provide anyway the entire visible information.

Let Δ be an alphabet (possibly infinite), and let $x \in \Delta^*$. Not assuming an interpretation of elements of Δ, what kind of structure can x be, what kind of information can x contain?

Consider $x = a.b.a.c.b.d$. The sequence x can be interpreted as a *total order* to_x of the occurrences of elements of Δ, as illustrated in Figure 2(a). By an occurrence of a we mean a pair (a, i), where i is a natural number indicating the occurrence.

Consider now the sequence $y = a.b.c.d$ and suppose that we have the additional information that the order between the occurrences of a, b, c, d does not matter. To express this, we introduce a partial operator $\langle\cdot\rangle$ which takes a *plain* sequence and removes the order of its elements. Sequences where each element of the alphabet occurs at most once are called plain. The set of all plain sequences over Δ is denoted by $Plain(\Delta)$. We can interpret $\langle\cdot\rangle$ as transforming a plain sequence into the corresponding set, for example $\langle y\rangle$" can be interpreted as $\{a, b, c, d\}$.

By mixing "$\langle\cdot\rangle$" with standard concatenation ".", we obtain sequences like $a.\langle a.c.b\rangle.\langle b.c\rangle.a.b$. Such sequences are used especially in concurrency theory. They are called *step-sequences* or *subset languages* ([18, 29]). They represent *weak* (or *stratified*) *partial orders* ([9, 18]). Figure 2(b) illustrates this relationship.

Formally step-sequences are constructed as sequences over the alphabet $Fin(2^{\Delta})$, where for every family of sets $\mathcal{X}$, $Fin(\mathcal{X}) = \{X \mid X \in \mathcal{X} \wedge X \text{ is finite}\}$. In this sense our $a.\langle a.c.b\rangle.\langle b.c\rangle.a.b$ corresponds to the sequence of sets: $\{a\}.\{a, c, b\}.\{b, c\}.\{a\}.\{b\}$.

From Szpirlajn theorem [9] it follows that every partial order corresponds uniquely to the set of all its total extensions. In particular every set $X = \{a_1, \ldots, a_n\}$ is a partial order with empty ordering relation, and it can be seen as a description of the set of all total order that can be built from the elements of X. Since finite total orders can be specified as sequences, the set $\{a_1, \ldots, a_n\}$ can be seen as a description of all *plain* sequences built from $a_1,\ldots,a_n$. For instance $\{a, b, c\}$ can be seen as a description of the set of sequences $\{a.b.c, a.c.b, b.a.c, b.c.a, c.a.b, c.b.a\}$.

In general a step-sequence can are be interpreted as a set of all sequences corresponding to all total extensions of the weak orders specified by the step-sequence. For instance the step-sequence $a.\langle b.a\rangle.c.\langle a.c\rangle$ defines the set of sequences: $\{a.b.a.c.a.c, a.a.b.c.a.c, a.b.a.c.c.a, a.a.b.c.c.a\}$. The set of sequences corresponding to the step-sequence from Figure 2(b) consists of 12 elements, including for instance $a.a.c.b.b.c.a.b$ and $a.b.c.a.c.b.a.b$

Formally, the set of step-sequences over Δ, denoted $\langle\Delta^*\rangle$, is the smallest set of sequences over $\Delta \cup \{\langle, \rangle\}$ such that:

- every $x \in \Delta^*$ is a step-sequence,
- if $x \in Plain(\Delta)$, then $\langle x\rangle$ is a step-sequence,
- if x and y are step-sequences, then $x.y$ is a step-sequence.

In addition to the above concatenation "." on step-sequences, we define *weak concatenation*, denoted by "$\smile$". Weak concatenation with an empty step-sequence is defined by:

$$x \smile \varepsilon = x \quad \text{and} \quad \varepsilon \smile y = y.$$

For non-empty step-sequences x and y, the idea of $x \smile y$ is to merge the last "step" of x with the first "step" of y. If the result of the concatenation of the last and first "step", respectively, is plain, then we have for instance $(a.b) \smile c = (a.c) \smile b = a.\langle b.c\rangle$, $\langle a.b\rangle \smile \langle c.d.e\rangle = \langle a.b.c.d.e\rangle$, and $(a.\langle a.b\rangle) \smile (\langle c.d\rangle.a) = a.\langle a.b.c.d\rangle.a$.

For non-plain step-sequences, "$\smile$" can be illustrated as follows: if $x = \langle a.b\rangle.c.\langle a.c\rangle$ and $y = \langle a.b\rangle.a.c$ then $x \smile y = \langle a.b\rangle.c.\langle a.b.c\rangle.a.c$, i.e. the last step of x, $\langle a.c\rangle$, is merged with the first step $\langle a.b\rangle$ of y. We would also like to write expressions like $(\langle a.b\rangle.c.\langle a.c\rangle) \smile (\langle a.b\rangle.a.c) = \langle a.b\rangle.c.(\langle a.c\rangle \smile \langle a.b\rangle).a.c$.

Formally, weak concatenation can be defined as follows. Since every non-empty step sequences x, y can be expressed as $x = x_1.\alpha$, $y = \beta.y_1$, where $\alpha = \langle t\rangle$ or $\alpha = a$, $\beta = \langle s\rangle$ or $\beta = b$, t and s are plain, $t \neq \epsilon$, $s \neq \epsilon$, $a, b \in \Delta$, we can define $x \smile y$ in this case by:

$$x \smile y = x_1.(\alpha \smile \beta).y_1$$

The interpretation of step-sequences is given by a mapping $sem : \langle\Delta^*\rangle \to 2^{\Delta^*}$. Let $set(x)$ denote the set of all elements of Δ, from which the string $x \in \Delta^*$ is built. For example $set(a.b.c.a.c) = \{a, b, c\}$. The mapping sem may be defined as follows:

1. $\forall x \in \Delta^* \,.\, sem(x) = \{x\}$,
2. $\forall x \in Plain(\Delta) \,.\, sem(\langle x\rangle) = \{y \in Plain(\Delta) \mid set(x) = set(y)\}$,
3. $\forall x, y \in \langle\Delta^*\rangle \,.\, sem(x.y) = sem(x).sem(y)$,

where "." in "$sem(x).sem(y)$" denotes the standard concatenation of sets of sequences (see [13]). For instance $sem(\langle a.b.c\rangle) = \{a.b.c, a.c.b, b.a.c, b.c.a, c.a.b, c.b.a\}$, $sem(a.\langle b.a\rangle.c.\langle a.c\rangle) = \{a.b.a.c.a.c$, $a.a.b.c.a.c$, $a.b.a.c.c.a$, $a.a.b.c.c.a\}$.

The two views of step sequences, sequences of sets and sets of sequences, are compatible in the sense that two step sequences $x, y \in \langle\Delta^*\rangle$ are equal, $x = y$, if and only if they are equal in their interpretations as sets of sequences, $sem(x) = sem(y)$.

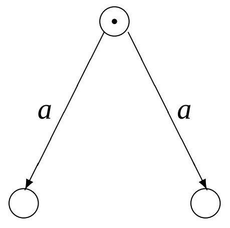
For all $x, y \in \langle\Delta^*\rangle$ we will say that x is a *prefix* of y if there is $z \in \langle\Delta^*\rangle$ such that $y = x.z$ or $y = x \smile z$. For every $t \in \langle\Delta^*\rangle$ and every $a \in \Delta$ we shall write $a \in t$ if a is contained in t. For instance $a \in b.\langle a.b\rangle$, and $a \notin b.b.c$. We use step-sequences to specify states of automata.

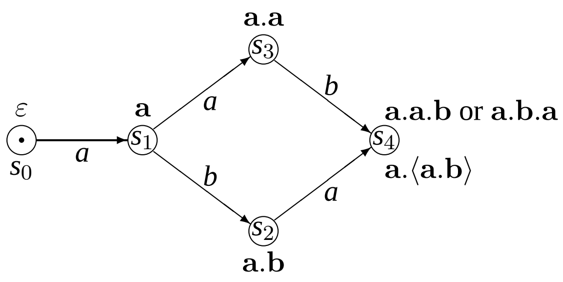
8 Trace Only Automata

Let $A = (\Delta, S, \varrho, s_0)$ be an automaton. We shall say that A has the *canonical trace property* (*ct-property*) if for every state $s \in S$ there is a trace $x_s \in \Delta^*$ such that $\varrho^*(x_s)(s_0) = \{s\}$. Not every automaton has ct-property and every transition-deterministic automaton has ct-property. The automaton from the left hand side of Figure 3 does not have ct-property (the automaton can then "generate" only two traces ϵ and a and it has three states). Frequently there is more than one x_s satisfying $\varrho^*(x_s)(s_0) = \{s\}$. For example for the automaton from Figure 1 and the state s_3 we have (s_1 is initial here) $\varrho^*(b{:}2.(a{:}2)^i)(s_1) = \{s_3\}$, for every $i \geq 0$.

If A has ct-property we may define a set of *canonical traces* [27]. A set of traces $CanTr \in \Delta^*$ is *canonical* if for every $s \in S$ there is exactly one $x_s \in CanTr$, its unique representation, such that $\varrho^*(x_s)(s_0) = \{s\}$. Automaton A is isomorphic to $A^{ct} = (\Delta, CanTr, \varrho^{ct}, x_{s_0})$, where $\varrho^{ct}(a)(x_s) = \{x_{s_1}, \ldots, x_{s_k}\} \iff \varrho(a)(s) = \{s_1, \ldots, s_k\}$.

Automata like A^{ct} are called *trace only automata* since their states are defined in terms of traces. Mealy machine counterparts of trace only automata are used extensively for the trace assertion method, e.g. [14, 16, 15, 24, 27, 33]. The problem is that using traces may frequently result in a kind of asymmetry which makes the specification less readable than expected. Consider the automaton on the right hand side of Figure 3. It occurs typically as part of a greater automaton. The state s_4 is unambiguously defined by two traces $a.a.b$ and $a.b.a$. Each of them can be chosen as a canonical one. If $a.a.b$ is chosen, then the canonical trace $x = a.a$ is a prefix of $a.a.b$, hence we have $\varrho^{ct}(b)(x) = \{x.b\}$. The canonical trace $y = a.b$ is not a prefix of $a.a.b$, so $\varrho^{ct}(a)(y) \neq \{y.a\}$. The asymmetry is induced by the choice of a canonical trace, the automaton itself is symmetrical, from the state s_1 we reach s_4 in two steps, using both a and b in any order, $a.b$ or $b.a$. Such asymmetry makes some specifications unnecessarily complex. The Unique Integer module is a classical example, but the problem occurs frequently in real modules


(a)



(b)

Figure 3: (a) Example of an automaton which does not have ct-property and (b) Example of asymmetry when canonical traces are used.

as well. The asymmetry disappears when step-traces are used to identify states. When the state s_4 is identified by $a.\langle a.b\rangle$, then both $a.b$ and $a.a$ are prefixes of $a.\langle a.b\rangle$ ($a.\langle a.b\rangle = (a.b) \smile a = (a.a) \smile b$).

For an automaton A with ct-property, we define the set $\mathcal{C} \subseteq \langle\Delta^*\rangle$ of *canonical step-traces*. Let $\bar{\varrho}$ be an extension of ϱ defined on the Cartesian product of the states and *plain traces*:

$$\bar{\varrho}(t) = \begin{cases} \varrho^*(x) & x \in sem(t) \wedge (\forall y, z \in sem(t) \ . \ \varrho^*(y) = \varrho^*(z)) \\ \emptyset & \text{otherwise} \end{cases}$$

Now, let $\mathcal{C}$ be any subset of $\langle\Delta^*\rangle$ satisfying:

1. $\forall t \in \mathcal{C}. \ \exists s_t \in S. \ \forall x \in sem(t) \ \varrho^*(x)(s_0) = \{s_t\}$,
2. $2\forall s \in S. \exists! t_s \in \mathcal{C}. \ \bar{\varrho}^*(t_s)(s_0) = \{s\}$.

The symbol $\exists!$ denotes “there exists exactly one”. Note that the ct-property implies the existence of (at least one) $\mathcal{C}$.

What if an automaton does not have the ct-property? First we must note that such a situation occurs rather seldom in practice. The Drunk Stack has the ct-property, Very Drunk Stack does not, but neither of them is a part of any real system. They were chosen to illustrate potential problems. But if the best and most readable model of a module is an automaton-like structure without ct-property, we can use a concept similar to *labeled transition system* [2]. In contrast to automata, each arrow in a transition system has a unique name. The elements of Δ attached to arrows in automata are called *labels* in transition systems. By “labeled transition system” we mean that each arrow has *two* attachments, a unique name, and a not necessarily unique label. We do not need each arrow to be unique, we need only the ct-property, so the following construction is proposed.

An *automaton with the alphabet of state constructors* is a tuple

$$A = (\Delta, \Upsilon, S, \varrho, s_0),$$

where Δ is the alphabet, Υ is the *state constructors* alphabet, S is the set of states, ϱ is the transition relation, $\varrho \in (\Delta \cup \Upsilon) \rightarrow S \leftrightarrow S$, and $s_0 \in S$ is the initial state. The transition relation ϱ must satisfy the following conditions:

1. $\forall \alpha \in \Upsilon, s \in S \, . \, |\varrho(\alpha)(s)| \leq 1$,
2. $\forall s_1, s_2 \in S \, . \, (\exists a \in \Delta \, . \, s_2 \in \varrho(a)(s_1) \;\Leftrightarrow\; \exists \alpha \in \Upsilon \, . \, \varrho(\alpha)(s_1) = \{s_2\})$.

The set $L(A) = \{x \in \Delta^* \mid \varrho^*(x)(s_0) \neq \emptyset\}$ describes the normal behavior of the module. The elements of Υ do not occur in $L(A)$. We *do not* assume $\Delta \cap \Upsilon = \emptyset$. The first condition says that A restricted to Υ is a transition-deterministic automaton. Hence the ct-property is guaranteed. The second condition guarantees that each arrow is marked by one element of Δ and one element of Υ. Since automata with state constructors alphabet do always have ct-property, their states can always be specified as canonical step-sequences.

Every automaton may be extended to an equivalent automaton with state constructor alphabet by simply defining $\Upsilon = \{(s, a, s') \mid s' \in \varrho(v)(s)\}$, and extending ϱ onto Υ by $\varrho(s, a, s')(s) = \{s'\}$. This construction results in a labeled transition system, and is of a very little use in practice, but is always possible.

9 Trace Assertion Specifications

Trace assertion specifications emerge when using canonical traces for the states of module specifications. More precisely, given a signature E, a trace assertion specification TA is a module specification

$$TA = (\Delta_E, \mathcal{C}, \varrho, t_0),$$

where $\mathcal{C} \subseteq \langle \Delta_E^* \rangle$ is a set of step-traces such that every step-trace t describes unambiguously one state, and this is the state the sequence $x \in sem(t)$ leads to:

$$\forall t \in \mathcal{C}, x \in sem(t) \, . \, \varrho^*(x)(t_0) = \{t\}.$$

For the Stack and Drunk Stack modules, $\mathcal{C}$ can just be the set of all sequences of type $PUSH(i_1)$. $PUSH(i_2)$. $\ldots$ $.PUSH(i_k)$, and for instance:

$$\begin{aligned}
&\varrho(TOP{:}4)(PUSH(5).PUSH(7).PUSH(4)) &&= \{PUSH(5).PUSH(7).PUSH(4)\} \\
&\varrho(TOP{:}8)(PUSH(5).PUSH(7).PUSH(4)) &&= \emptyset \\
&\varrho(PUSH(5))(PUSH(5).PUSH(7).PUSH(4)) &&= \{PUSH(5).PUSH(7).PUSH(4).PUSH(5)\}
\end{aligned}$$

The access program called POP behaves differently in Stack than in Drunk Stack, for example:

$$\varrho(POP)(PUSH(5).PUSH(7).PUSH(4)) = \{PUSH(5).PUSH(7)\}$$

for Stack, while for Drunk Stack:

$$\varrho(POP)(PUSH(5).PUSH(7).PUSH(4), POP) = \{PUSH(5).PUSH(7), PUSH(5)\}.$$

For the Unique Integer module, the set $\mathcal{C}$ can be defined as the set of all step-sequences $\langle GET{:}i_1.GET{:}i_2. \ldots .GET{:}i_k\rangle$, where $i_j = i_k \Leftrightarrow j = k$, and for instance:

$$\begin{aligned}
\varrho(GET{:}7)(GET{:}3.GET{:}6.GET{:}9) &= \{\langle GET{:}3.GET{:}6.GET{:}9.GET{:}7\rangle\} \\
&= \{\langle GET{:}3.GET{:}6.GET{:}7.GET{:}9\rangle\} \\
\varrho(GET{:}3)(GET{:}3.GET{:}6.GET{:}9) &= \emptyset
\end{aligned}$$

The Very Drunk Stack cannot be modeled (in a natural way) by *TA* as defined above.

Given a trace assertion specification *TA*, we define the competence function $\kappa : \mathcal{C} \times \Sigma \to Bool = \{0,1\}$, in the following way:

$$\kappa(t,a) = \begin{cases} 0 \text{ if } \forall d \in \mathcal{O} \ . \ \varrho(a{:}d)(t) = \emptyset \\ 1 \text{ if } \exists d \in \mathcal{O} \ . \ \varrho(a{:}d)(t) \neq \emptyset \end{cases}$$

The notion of competence function follows from [24]. It defines what is a misuse. If $\kappa(t,a) = 0$, then the use of a at the state described by t is a misuse. For both Stack and Drunk Stack we have: $\kappa(\varepsilon, POP) = \kappa(\varepsilon, TOP) = 0$, and $\kappa(t_{full}, PUSH(d)) = 0$, if t_{full} represents the full stack. Otherwise $\kappa(t,a) = 1$. For the Unique Integer, $\kappa(t, GET) = 0$ only if t represents the state where all available integers are used.

Let $\pi : \langle\Delta^*\rangle \to \langle\Sigma^*\rangle$ be a projection mapping, for any $a{:}d \in \Delta$ and $x,y \in \langle\Delta^*\rangle$ defined by:

$$\pi(\varepsilon) = \varepsilon, \qquad \pi(a{:}d) = a, \qquad \pi(x.y) = \pi(x).\pi(y), \qquad \pi(\langle x\rangle) = \langle\pi(x)\rangle.$$

For example $\pi(a_1{:}d_1.a_2{:}d_2.a_3{:}d_3.a_4{:}d_4) = a_1.a_2.a_3.a_4$, and $\pi(a{:}3.\langle a{:}2.b{:}2\rangle.\langle a{:}3.a{:}2\rangle) = a.\langle a.b\rangle.\langle a.a\rangle$.

A trace assertion specification *TA* is *output independent* if for every $x,y \in L(TA)$,

$$x = y \iff \pi(x) = \pi(y),$$

otherwise it is *output dependent*. If *TA* is output independent then π can be interpreted as a one-to-one function, so π^{-1} is a function on $\pi(L(TA))$.

Both Stack and Drunk Stack are output independent while Unique Integer is not. Note that in [27, 33] and others the output independent trace assertion specifications are called deterministic while output dependent are called non-deterministic.

In our terminology, both Stack and Unique Integer are transition deterministic, while Drunk Stack is not. Transition determinism does not imply output independence and output independence does not imply transition determinism. Unique Integer is transition deterministic but output dependent, Drunk Stack is not transition deterministic but output independent.

10 Mealy Form of Trace Assertion Specifications

If *TA* is output independent, it may be represented as a kind of a Mealy machine, with a separate specification of the output function. Most trace assertion models in the literature are based on Mealy machines. We think that, in general, the automata concept is better, but for the output independent *TA*'s the Mealy model also leads to equally readable specification. It also helps to explain the relationship with algebraic specifications (see Chapter 16).

Lemma 10.1 *If TA is output independent then, for all $t \in \mathcal{C}$, $a \in \Sigma$, and all $d \in \mathcal{O}$,*

$$\varrho(a{:}d)(t) \neq \emptyset \ \Rightarrow \ (\forall d' \neq d \ . \ \varrho(a{:}d')(t) = \emptyset).$$

Proof. Suppose that there are $d, d' \in \Delta^*$, $a \in \mathcal{C}$, such that $d \neq d'$, and $\varrho(a{:}d)(t) \neq \emptyset$, $\varrho(a{:}d')(t) \neq \emptyset$. Hence for all $x \in sem(t)$, $x.a{:}d \neq x.a{:}d'$, while $\pi(x.a{:}d) = \pi(x).a = \pi(t.a{:}d')$, a contradiction. ■

Lemma 10.1 says that for output independent *TA*, for every $t \in \mathcal{C}$, $a \in \Sigma$, there exists *at most* one $d \in \mathcal{O}$ such that $\varrho(a{:}d)(t)$ is not empty.

Given an *output independent* trace assertion specification, we define the mapping $\delta : \Sigma \to \pi(\mathcal{C}) \leftrightarrow \pi(\mathcal{C})$, the *calls only transition function*, as

$$\delta(a)(\pi(t)) = \pi(\varrho(a{:}d)(t))$$

and the mapping $v : \Sigma \to \pi(\mathcal{C}) \to \mathcal{O} \cup \{nil\}$, the *output value function* as follows:

$$v(\pi(t), a) = \begin{cases} d & \text{if } \exists d \in \mathcal{O} \ . \ \varrho(a{:}d)(t) \neq \emptyset \\ nil & \text{if } \forall d \in \mathcal{O} \ . \ \varrho(a{:}d)(t) = \emptyset \end{cases}$$

Lemma 10.1 guarantees the well-definedness of δ and v.

Proposition 10.2 *If TA is output independent and deterministic then for all $t, s \in \mathcal{C}$, $a{:}d \in \Delta$:*

$$s \in \varrho(a{:}d)(t) \quad \Longleftrightarrow \quad \pi(s) \in \delta(a)(\pi(t)) \wedge v(a)(\pi(t)) = d.$$

Proof. ($\Rightarrow$) From the definitions of δ and v.
($\Leftarrow$) Suppose $s \notin \varrho(a{:}d)(t)$. We have to consider two cases.
Case 1. $\varrho(a{:}d)(t) = \emptyset$. From the definition of v, we have $v(a)(\pi(t)) = nil \neq d$.
Case 2. $\varrho(a{:}d)(t) \neq \emptyset$. Then $\delta(a)(\pi(t)) = \pi(\varrho(a{:}d)(t))$. If $\pi(s) \in \delta(a)(\pi(t))$ then there exists $s' \in \mathcal{C}$ such that $\pi(s) = \pi(s')$, a contradiction, since *TA* is output independent. ■

Proposition 10.2 guarantees that for output independent *TA*'s, the mapping ϱ is completely defined by δ and v.

The Stack is output independent, so instead of

$$\varrho(TOP{:}4)(PUSH(5).PUSH(7).PUSH(4)) = \{PUSH(5).PUSH(7).PUSH(4)\},$$

one can write equivalently, as commonly used in the existing literature:

$$\delta(TOP)(PUSH(5).PUSH(7).PUSH(4)) = \{PUSH(5).PUSH(7).PUSH(4)\},$$
$$v(TOP)(PUSH(5).PUSH(7).PUSH(4)) = 4.$$

Proposition 10.2 allows to represent any output independent trace assertion specification in an equivalent form, which is called the *Mealy form*. Formally the Mealy form of *TA* is defined by:

$$TA^{Mealy} = (\Delta_E, \pi(\mathcal{C}), \delta, v, t_0).$$

For every trace assertion specification *TA*, not necessary output independent, an explicate value function $v_\varrho : \Sigma \to \mathcal{C} \leftrightarrow \mathcal{O}$ can be defined as $v_\varrho(a)(t) = \{d \mid \varrho(a{:}d)(t) \neq \emptyset\}$. But v_ϱ differs from v. The function v can only be defined for an output independent *TA*, occurs together with δ and cannot be derived form δ. The mapping v_ϱ is redundant, it is derived from ϱ.

For the Unique Integer module which is output *dependent* one may just write

$$\begin{aligned}
&\varrho(\langle GET{:}3.GET{:}6.GET{:}9\rangle, GET{:}7) = \{\langle GET{:}3.GET{:}6.GET{:}9.GET{:}7\rangle\},\\
&\varrho(\langle GET{:}3.GET{:}6.GET{:}9\rangle, GET{:}i) \;= \emptyset \text{ if } i \in \{3,6,9\},
\end{aligned}$$

or one may write equivalently:

$$\begin{aligned}
&\varrho(\langle GET{:}3.GET{:}6.GET{:}9\rangle, GET{:}7) = \{\langle GET{:}3.GET{:}6.GET{:}7.GET{:}9\rangle\},\\
&\varrho(\langle GET{:}3.GET{:}6.GET{:}9\rangle, GET{:}i) \;= \emptyset \text{ if } i \in \{3,6,9\},\\
&v_\varrho(\langle GET{:}3.GET{:}6.GET{:}9\rangle, GET) \;= \{i \mid i \notin \{3,6,9\}\}.
\end{aligned}$$

The second way is longer and, in our opinion, does not increase readability. It is used in [33] and others.

All the concepts introduced so far do not allow to model Very Drunk Stack in a natural way. The reason is that in this case the natural states of the stack, i.e. the sequences of integers, cannot be unambiguously described by the sequences of call-responses. The solution we suggested in Section 2 is to introduce a state constructor $push1(i)$ to describe the stack states.

11 Trace Assertion Specifications with State Constructors

Given a signature E, a *trace assertion specification with state constructors* is an automaton

$$\Upsilon TA = (\Delta_E, \Upsilon, \mathcal{C}, \varrho, t_0),$$

with the alphabet Υ of state constructors where, as for trace assertion specifications, $\mathcal{C} \subseteq \langle \Upsilon^* \rangle$ is the set of *canonical step-traces*:

$$\forall t \in \mathcal{C}, x \in sem(t) \;.\; \varrho^*(x)(t_0) = \{t\}.$$

We do not assume $\Delta_E \cap \Upsilon = \emptyset$, although it may often happen. The elements of $\Upsilon \setminus \Delta_E$ are *invisible* (abstract).

For the Very Drunk Stack module, the set Υ is the set of all abstract invisible calls $push1(i)$, where i is any available integer, and $\mathcal{C}$ is the set of all sequences $push1(i_1).\ \ldots\ .push1(i_k)$. In this case we have $\Delta_E \cap \Upsilon = \emptyset$. For instance

$$\begin{aligned}
&\varrho(TOP{:}4)(push1(5).push1(7).push1(4)) &&= \{push1(5).push1(7).push(4)\}\\
&\varrho(TOP{:}8)(push1(5).push1(7).push1(4)) &&= \emptyset\\
&\varrho(PUSH(5))(push1(5).push1(7).push1(4)) &&= \{t_1, t_2\}\\
&\varrho(POP)(push1(5).push1(7).push1(4)) &&= \{push1(5).push1(7), push1(5)\},
\end{aligned}$$

where

$$\begin{aligned}
t_1 &= push1(5).push1(7).push1(4).push1(5),\\
t_2 &= push1(5).push1(7).push1(4).push1(5).push1(5).
\end{aligned}$$

A trace assertion specification with state constructors ΥTA defines the following normal behavior

$$L(\Upsilon TA) = \{x \in \Delta^* \mid \varrho^*(x)(t_0) \neq \emptyset\}.$$

Note that Υ is not involved in $L(\Upsilon TA)$. The output independent ΥTA, the Mealy form of an output independent ΥTA, and the competence function κ are defined analogously as for TA.

Introducing invisible state constructors is clearly *against* the philosophy of the trace assertion method as formulated in [3]. One of the advantages claimed in [3] was no need for hidden functions to specify modules with delays. The algebraic specifications of those modules have required hidden functions. On the other hand, what we really want is to specify the visible behavior of a module in the most easy and readable yet precise way. The states are auxiliary concepts, and the invisible calls seem to serve well as the state constructors.

12 Enhancements and Exceptional Behavior

Large specifications are best developed and presented in a number of steps of increasing complexity. In particular, we suggest that the first step describes the normal behavior and exceptional behavior is added in the second (or later) step. The second step can be seen as an *enhancement* of the first step, in the sense that it additional behavior is specified while the original is preserved.

Let us take the stack module and a trace $t = PUSH(i_1).PUSH(i_2).\ldots.PUSH(i_n)$. Suppose that the stack has a bound n, i.e. t is a state of the full stack, and consider the trace:

$$t.PUSH(i_{n+1}) = PUSH(i_1).PUSH(i_2).\ \ldots\ .PUSH(i_n).PUSH(i_{n+1}).$$

Since we cannot prevent such an access program call to occur, the question arises what behavior this trace describes. Defining the transition relation to be empty in this case allows nontermination. Alternatively, we can specify that $PUSH(i_{n+1})$ should be ignored or that it replaced the previous top element.

However, in any case the state structure of the module is independent of its exceptional behavior. All states of the stack are entirely defined by its normal behavior.

Let $TA = (\Delta_E, \mathcal{C}, \varrho, t_0)$ be a trace assertion specification. If $\kappa(t,a) = 0$ then for all $d \in \mathcal{O}$, we have $\varrho(a{:}d)(t) = \emptyset$, which means that a at t is a misuse and it does not generate any normal behavior. In principle, an enhancement of TA consists in defining new ϱ' such that $\varrho'(a{:}d)(t) \neq \emptyset$ when $\kappa(t,a) = 0$. It is a structure complimentary to TA. Formally, an enhancement $enh(TA)$ of TA is a triple

$$enh(TA) = (\Delta_{E'}, \mathcal{C}', \varrho'),$$

where: $\Delta_{E'}$ is an *enhanced call-response alphabet,* $\mathcal{C}' \subseteq \langle \Delta^*_{E'} \rangle$ is an *enhanced set of canonical traces,* $\varrho' \in \Delta_{E'} \to \mathcal{C}' \leftrightarrow \mathcal{C}'$ is an *enhanced transition relation,* which is defined only if the transition relation of TA is not defined,

$$\forall t \in \mathcal{C}\ .\ \forall a \in \Sigma_E\ .\ \kappa'(t,a) = 1 \Rightarrow \kappa(t,a) = 0,$$

where κ and κ' are the competence functions of TA and $enh(TA)$, respectively.

The enhancement $enh(TA)$ is called *plain* if $\Delta_{E'} \subseteq \Delta_E$, and $\mathcal{C}' \subseteq \mathcal{C}$. Non-plain $enh(TA)$ means that there are some special error recovery states and some separate error recovery procedure. We shall not consider such examples in this paper.

For Stack and Drunk Stack a plain enhancement can be defined by:

$$\varrho'(POP)(\varepsilon) = \varrho'(TOP)(\varepsilon) = \{\varepsilon\} \qquad \text{and} \quad \varrho'(PUSH(i))(t_{full}) = \{t_{full}\}$$

where t_{full} is the canonical step-trace corresponding to the full stack, and $\varrho'(a{:}d)(t) = \emptyset$ for the rest of $t, a,$ and d. For the Unique Integer module the enhancement can be defined by:

$$\varrho'(GET{:}nil)(t_{all}) = \{t_{all}\}$$

where t_{all} is the canonical trace corresponding to the state where all available integers are used up, and $\varrho'(a{:}d)(t) = \emptyset$ for all other t, a, and d.

Enhancements describing exceptional behavior are typically output independent, hence can be represented in a Mealy form $(\Delta_{E'}, \pi(\mathcal{C}'), \delta', v')$. The definition is practically identical as for output independent *TA*'s. The only difference is that the enhancements do not possess initial step-traces.

For Stack and Drunk Stack the plain form of an enhancement can be defined by:

$$\delta'(POP)(\varepsilon) = \delta'(TOP)(\varepsilon) = \{\varepsilon\}, \qquad \delta'(PUSH(d))(t_{full}) = \{t_{full}\}, \qquad v'(TOP)(t_{full}) = nil,$$

and $\delta'(a)(t) = \emptyset$ for the rest of t and a.

We obtain the *enhanced trace assertion specification ETA* by taking the composition (union) of *TA* and the enhancement $enh(TA)$: The full specification is just a union of *TA* and $enh(TA)$, $ETA = TA \cup enh(TA)$, i.e.

$$ETA = (\Delta_E \cup \Delta_{E'}, \mathcal{C} \cup \mathcal{C}', \varrho \cup \varrho', t_0).$$

Hence $\varrho^+ = \varrho \cup \varrho'$ satisfies $\varrho^+ \in (\Delta \cup \Delta_E) \rightarrow (\mathcal{C} \cup \mathcal{C}') \leftrightarrow \mathcal{C} \cup \mathcal{C}'$, and for all $t \in \mathcal{C} \cup \mathcal{C}', a \in \Delta_E \cup \Delta_{E'}, d \in \mathcal{O} \cup \mathcal{O}'$,

$$\varrho^+(a{:}d)(t) = \begin{cases} \varrho(t, a{:}d) & \text{if } t \in \mathcal{C} \wedge \kappa(t,a) = 1 \\ \varrho'(t, a{:}d) & \text{otherwise} \end{cases}$$

Proposition 12.1 *For any plain enhancement $enh(TA)$:*

$$TA \leq TA \cup enh(TA)$$

Proof. Since $enh(TA)$ is plain, we have that $ETA = TA \cup enh(TA) = (\Delta_E, \mathcal{C}, \varrho \cup \varrho', t_0)$. We apply Proposition 5.1 with $R = Id$. Refinement follows immediately from the above observation that ϱ^+ defines additional behavior only if $\kappa(t,a) = 0$. ■

For every output independent *ETA* we can standardly built its Mealy form ETA^{Mealy}. In a very similar way we may define an enhancement for the trace assertion specification with state constructors as a composition of a trace assertion specification with state constructors and its enhancement.

13 Specification Format

To be useful in practice, the trace assertion technique must provide some specification formats. Two such formats are described and later used. Any trace assertion specification in the standard form consists of four sections: *Syntax, Canonical Step-trace Definition, Trace Assertions* and *Dictionary*. A trace assertion specification in the *Mealy form* consists of five sections: *Syntax, Canonical Step-trace Definition, Trace Assertions, Output Values* and *Dictionary*.

In the Mealy form the *Syntax* section is just a table which specifies for each module access-program name $f \in E$, the possible inputs $input(f)$ and by the number of arguments each program takes and the type of each argument, and the possible outputs $output(f)$ by the type of each return value. In the standard form it also specifies call-response formats for all access programs.

In the *Canonical Step-trace Definition* section, the predicate *canonical* and the initial canonical step-trace are defined. In general this could be a complex definition with a tabular notation involved (c.f. [27, 33]). However, in the majority of (well thought of) cases this is a relatively simple formula. The convention

$$[e(x_i)]_{i=j}^{k}$$

as a shorthand for $e(x_j).e(x_{j+1}). \;\dots\; .e(x_k)$ and ε if $k < j$, is often used.

In the standard form the Trace Assertions section is a sequence of trace assertions of the form

$$\varrho(a{:}d)(t) = \{t_1, \dots, t_k\}$$

for all calls a defined in the Syntax section. The traces $t, t_1, \dots, t_k$ are the canonical step-traces. Since ϱ is a total function it must be defined for every possible a and d. The convention that empty relations or sets, respectively, are specified by omission is used. If for particular values of t, a and d, the value of the function $\varrho(a{:}d)(t)$ does not appear in the Trace Assertions section this means that $\varrho(a{:}d)(t) = \emptyset$. In the Mealy form the Trace Assertions section is a sequence of trace assertions of the form $\delta(a)(t) = \{t_1, \dots, t_k\}$.

To specify transition deterministic trace assertions the following tabular notation is used[2].

$\varrho(a{:}d)(t) =$

Conditions	Trace Patterns	Equivalence
*condition*1	*pattern*1(t)	*this_c'*
.....	.....	.....

The column *Equivalence* defines the canonical step-trace t' such that $\varrho(a{:}d)(t) = \{t'\}$. Since t here is a variable, t' could be different for different t, the columns *Conditions* and *Trace Patterns* are used to specify all different cases. The column *Trace Patterns* contains appropriate patterns (or their characteristic predicates) for t, while the column *Conditions* contains predicates on the trace and argument variables. The first row above should be read *if condition*1 *and pattern*1(t) *then* $\varrho(a{:}d)(t) = \{\mathit{this\_c}'\}$. The columns *Conditions* and *Trace Patterns* can be omitted if not needed. The empty cells in those columns denote the predicate *true*.

For trace assertions which are not transition deterministic the tabular notation is slightly different, namely:

$\varrho(a{:}d)(t) =$

Conditions	Trace Patterns	Clusters			
*condition*1	*pattern*1(t)	$t_{1,1}$	$t_{1,2}$	$t_{1,3}$	$t_{1,4}$
*condition*2	*pattern*2(t)	$t_{2,1}$		$t_{2,2}$	
.....	.....	.....			

In this case the rows should be read as follows:

*if condition*1 *and pattern*1(t) *then* $\varrho(a{:}d)(t) = \{t_{1,1}, t_{1,2}, t_{1,3}, t_{1,4}\}$,

*if condition*2 *and pattern*2(t) *then* $\varrho(a{:}d)(t) = \{t_{2,1}, t_{2,2}\}$,

etc. Since in this case the canonical step-traces do not represent equivalence classes but some clusters of traces the third column has now the name *Clusters*.

For the Mealy form we have also the *Output Values* section, which defines the value function v. A similar tabular notation is used, in this case a table consists of the columns *Conditions, Trace Patterns* and *Value*. The *nil* values are specified by omission.

[2] See [17, 19, 26] for details on tabular notation.

Syntax of Access Programs

Name	Argument	Value	Call-Response Forms
POP			POP:nil
PUSH	*integer*		$PUSH(d)$:nil
TOP		*integer*	TOP:d

Canonical Step-traces

$canonical(t) \quad \Leftrightarrow \quad t = [PUSH(d_i)]_{i=1}^{n} \wedge 0 \leq n \leq size$

$t_0 = \varepsilon$

Trace Assertions

$\varrho(POP)(t) =$

Trace Patterns	Equivalence
$t = s.PUSH(d)$	s
% $\quad t = \varepsilon$	ε

$\varrho(PUSH(d))(t) =$

Condition	Equivalence
$length(t) < size$	$t.PUSH(d)$
% $\quad length(t) = size$	t

$\varrho(TOP$:$d)(t) =$

Condition	Trace Patterns	Equivalence
	$t = s.PUSH(d)$	t
% $\quad d = nil$	$t = \varepsilon$	ε

Dictionary

size : the size of the stack

$length(t)$: the length of the trace t

Figure 4: Enhanced Trace Assertion Specification for Stack Module

The *Dictionary* section provides the definitions of the terms, auxiliary functions, types and other structures that are used in the body of the specification. The Dictionary section is rather short for simple examples.

The format for *enhanced* trace assertion specification is basically the same as the described above. The only difference is that new rows that correspond to the enhancement are added. We use the convention that all the rows added by the enhancement are marked by the symbol “%” at the beginning.

Figures 4, 5, 6, 7, 8 present various forms of trace assertion specifications of the Stack, Drunk Stack and Very Drunk Stack modules in the specification format described above.

Syntax of Access Programs

Name	Argument	Value
POP		
PUSH	*integer*	
TOP		*integer*

Canonical Step-traces

$canonical(t) \quad \Leftrightarrow \quad t = [PUSH(d_i)]_{i=1}^{n} \wedge 0 \leq n \leq size$

$t_0 = \varepsilon$

Trace Assertions

$\delta(POP)(t) =$

Trace Patterns	Equivalence
$t = s.PUSH(d)$	s
% $t = \varepsilon$	ε

$\delta(PUSH(d))(t) =$

Condition	Equivalence
$length(t) < size$	$t.PUSH(d)$
% $length(t) = size$	t

$\delta(TOP)(t) =$

Equivalence
t

or $\delta(TOP)(t) =$

Trace Patterns	Equivalence
$t \neq \varepsilon$	t
% $t = \varepsilon$	t

Values

$v(TOP)(t) =$

Trace Patterns	Value
$t = s.PUSH(d)$	d
% $t = \varepsilon$	*nil*

Dictionary

size : the size of the stack

$length(t)$: the length of the trace t

Figure 5: Mealy Form of the Enhanced Trace Assertion Specification for Stack Module

Syntax of Access Programs

Name	Value	Call-Response Forms
GET	*integer*	*GET:d*

Canonical Step-traces

$canonical(t) \quad \Leftrightarrow \quad t = \langle [GET{:}d_i]_{i=1}^{n} \rangle \wedge 0 \leq n \leq size \wedge (d_i = d_j \Leftrightarrow i = j)$

$t_0 = \varepsilon$

Trace Assertions

$\varrho(GET{:}d)(t) =$

Condition	Equivalence
$length(t) < limit \wedge GET{:}d \notin t$	$t \smile GET{:}d$
% $\quad length(t) = limit \wedge d = nil$	t

Dictionary

limit : the number of available integers $limit = maxinteger - mininteger + 1$

maxinteger : the maximum available integer

mininteger : the minimum available integer

$length(t)$: the length of the trace t

Figure 6: Enhanced Trace Assertion Specification for Unique Integer Module

14 Refining Modules

We illustrate the refinement of trace assertion specifications by showing that Drunk Stack is refined by Stack. From the Trace Assertion section in Figure 4 we get following transition relation ϱ_S for Stack:

$$\begin{aligned}
\varrho_S(POP)(t,t') &\Leftrightarrow \exists d, s \;.\; (t = s.PUSH(d) \wedge t' = s) \vee (t = \varepsilon \wedge t' = \varepsilon) \\
&\Leftrightarrow \exists d \;.\; t = t'.PUSH(d) \vee (t = \varepsilon \wedge t' = \varepsilon), \\
\varrho_S(PUSH(d))(t,t') &\Leftrightarrow (length(t) < size \wedge t' = t.PUSH(d)) \vee (length(t) = size \wedge t' = t), \\
\varrho_S(TOP{:}d)(t,t') &\Leftrightarrow \exists s \;.\; (t = s.PUSH(d) \wedge t' = t) \vee (d = nil \wedge t = \varepsilon \wedge t' = \varepsilon).
\end{aligned}$$

We transform ϱ into $\tilde{\varrho}$ such that $(a,t)\ \tilde{\varrho}\ (d,t') \Leftrightarrow t\ (\varrho(a{:}d))\ t'$ (see Section 5):

$$\begin{aligned}
\tilde{\varrho}_S(POP,t)(nil,t') &\Leftrightarrow \varrho_S(POP)(t,t'), \\
\tilde{\varrho}_S(PUSH(d),t)(nil,t') &\Leftrightarrow \varrho_S(PUSH(d))(t,t'), \\
\tilde{\varrho}_S(TOP,t)(d,t') &\Leftrightarrow \varrho_S(TOP{:}d)(t,t').
\end{aligned}$$

Syntax of Access Programs

Name	Argument	Value	Call-Response Forms
POP			*POP*:*nil*
PUSH	*integer*		*PUSH*(*d*):*nil*
TOP		*integer*	*TOP*:*d*

Canonical Step-traces

$canonical(t) \quad \Leftrightarrow \quad t = [PUSH(d_i)]_{i=1}^{n} \wedge 0 \leq n \leq size$

$t_0 = \varepsilon$

Trace Assertions

$\varrho(POP)(t) =$

Trace Patterns	Clusters	
$t = PUSH(d)$	ε	
$t = s.PUSH(d_1).PUSH(d_2)$	$s.PUSH(d_1)$	s
% $t = \varepsilon$	ε	

$\varrho(PUSH(d))(t) =$

Condition	Equivalence
$length(t) < size$	$t.PUSH(d)$
% $length(t) = size$	t

$\varrho(TOP{:}d)(t) =$

Condition	Trace Patterns	Equivalence
	$t = s.PUSH(d)$	t
% $d = nil$	$t = \varepsilon$	ε

Dictionary

size : the size of the stack

$length(t)$: the length of the trace t

Figure 7: Enhanced Trace Assertion Specification for Drunk Stack Module

Syntax of Access Programs

Visible Name	Abstract Name	Argument	Value	Call-Response Forms
POP				*POP*:*nil*
PUSH		*integer*		$PUSH(d)$:*nil*
TOP			*integer*	*TOP*:*d*
	$push1$	*integer*		$push1(d)$

Canonical Step-traces

$canonical(t) \quad \Leftrightarrow \quad t = [push1(d_i)]_{i=1}^{n} \wedge 0 \leq n \leq size$

$t_0 = \varepsilon$

Trace Assertions

$\varrho(POP)(t) =$

Trace Patterns	Clusters	
$t = push1(d)$	ε	
$t = s.push1(d_1).push1(d_2)$	$s.push1(d_1)$	s
% $t = \varepsilon$	ε	

$\varrho(PUSH(d))(t) =$

Condition	Cluster	
$length(t) < size - 1$	$t.push1(d).push1(d)$	$t.push1(d)$
$length(t) = size - 1$	$t.push1(d)$	
% $lenght(t) = size$	t	

$\varrho(TOP{:}d)(t) =$

Condition	Trace Patterns	Equivalence
	$t = s.PUSH(d)$	t
% $d = nil$	$t = \varepsilon$	ε

Dictionary

size : the size of the stack

$length(t)$: the length of the trace t

Figure 8: Enhanced Trace Assertion Specification for Very Drunk Stack Module

From the Trace Assertion section in Figure 7 we get the following transition relation ϱ_{DS} for Drunk Stack:

$$\begin{aligned}
\varrho_{DS}(POP)(t,t') &\Leftrightarrow \exists d, d_1, d_2, s \,.\, (t = PUSH(d) \wedge t' = \varepsilon) \vee \\
&\quad (t = s.PUSH(d_1).PUSH(d_2) \wedge (t' = s.PUSH(d_1) \vee t' = s) \vee \\
&\quad (t = \varepsilon \wedge t' = \varepsilon) \\
&\Leftrightarrow \exists d, d_1, d_2 \,.\, t = t'.PUSH(d) \vee t = t'.PUSH(d_1).PUSH(d_2) \vee, \\
&\quad (t = \varepsilon \wedge t' = \varepsilon) \\
\varrho_{DS}(PUSH(d))(t,t') &\Leftrightarrow (length(t) < size \wedge t' = t.PUSH(d)) \vee (length(t) = size \wedge t' = t), \\
\varrho_{DS}(TOP{:}d)(t,t') &\Leftrightarrow \exists s \,.\, (t = s.PUSH(d) \wedge t' = t) \vee (d = nil \wedge t = \varepsilon \wedge t' = \varepsilon).
\end{aligned}$$

We transform ϱ_{DS} into $\tilde{\varrho}_{DS}$:

$$\begin{aligned}
\tilde{\varrho}_{DS}(POP,t)(nil,t') &\Leftrightarrow \varrho_{DS}(POP)(t,t'), \\
\tilde{\varrho}_{DS}(PUSH(d),t)(nil,t') &\Leftrightarrow \varrho_{DS}(PUSH(d))(t,t'), \\
\tilde{\varrho}_{DS}(TOP,t)(d,t') &\Leftrightarrow \varrho_{DS}(TOP{:}d)(t,t').
\end{aligned}$$

For showing simulation between Drunk Stack and Stack, we have to find a relation R between the canonical traces of Drunk Stack and those of Stack. Let $subseq(x,y)$ be a relation between sequences x and y which holds if elements of x occur in the same order in y, i.e. $subseq$ is the smallest relation such that for any sequences x, y and element a:

$$subseq(\varepsilon,\varepsilon) \quad \text{and} \quad subseq(x,y) \Rightarrow subseq(x.a, y.a) \quad \text{and} \quad subseq(x,y) \Rightarrow subseq(x, y.a).$$

Intuitively, the canonical traces of Drunk Stack correspond to those of Stack with some $PUSH(d)$ elements interspersed. Hence we define:

$$R(t,t') \Leftrightarrow subseq(t',t).$$

The first condition for Stack to simulate Drunk Stack using R is that the initial trace $t_0 = \varepsilon$ of Drunk Stack and $t_0 = \varepsilon$ of Stack are in relation R, which holds trivially. The second condition is $\tilde{\varrho}_{DS} \sqsubseteq_{Id \times R} \tilde{\varrho}_S$, which is defined as:

$$\tilde{\varrho}_{DS}(a,t) \neq \emptyset \Rightarrow ((Id \times R) \circ \tilde{\varrho}_S)(a,t) \subseteq (\tilde{\varrho}_{DS} \circ (Id \times R))(a,t) \wedge ((Id \times R) \circ \tilde{\varrho}_S)(a,t) \neq \emptyset,$$

where a are all the calls of Drunk Stack and Stack, and t ranges over all canonical trances of Drunk Stack. We consider the cases $a = POP$, $a = PUSH(d)$, and $a = TOP$ separately. For $a = POP$, we have that $\tilde{\varrho}_{DS}(POP,t) \neq \emptyset$ for any canonical trace t and similarly $\tilde{\varrho}_s(POP,t) \neq \emptyset$. As R is a total relation, it is easy to see that $((Id \times R) \circ \tilde{\varrho}_S)(POP,t) \neq \emptyset$ for any canonical trace t. Hence above condition simplifies in this case to:

$$((Id \times R) \circ \tilde{\varrho}_S)(POP,t) \subseteq (\tilde{\varrho}_{DS} \circ (Id \times R))(POP,t),$$

which is equivalent to:

$$\begin{aligned}
&(\exists t' \,.\, subseq(t',t) \wedge (\exists d \,.\, t' = t''.PUSH(d) \vee (t' = \varepsilon \wedge t'' = \varepsilon))) \\
\Rightarrow\ &(\exists t' \,.\, (\exists d, d_1, d_2 \,.\, t = t'.PUSH(d) \vee t = t'.PUSH(d_1).PUSH(d_2) \vee (t = \varepsilon \wedge t' = \varepsilon)) \\
&\qquad \wedge subseq(t'',t')),
\end{aligned}$$

for all t''. This holds according to the rules of logic and above definition of *subseq*. The cases $a = PUSH(d)$ and $a = TOP$ follow similarly. In total, this establishes Drunk Stack $\sqsubseteq_R$ Stack, which according to Proposition 5.1 implies Drunk Stack $\leq$ Stack.

Trace assertion specifications like Stack can be further refined into modules with a "more concrete" state space. For example, a Stack implementation could use an array A and integer N, related to the canonical traces of Stack by

$$R(t, A, N) \Leftrightarrow N = length(t) \wedge t = A[1..N],$$

where $A[1..N]$ selects the subsequence of A with the first N elements. Such refinement steps can be carried out in a standard way, e.g. [10, 21]. However, this establishes a link between abstract trace assertion specifications and efficient implementations.

15 Multi-Object Modules

In practical applications, it is not unusual that a module is designed to implement several independent homogeneous objects. For example in some applications, one may need to design a (multi-object) stack module that implements two or, in general, any number of stacks, plus for instance the stack *concatenation* operation. The module may be *self-initializing*, i.e. the first use of $PUSH(stack\_name, i)$ creates a stack *stack_name* or may require object generator like $new(stack\_name)$. A natural way of modeling such modules is to define the global states as sets of states of individual modules, with the empty set as the initial state. We already know how to specify individual states (by canonical step-traces) and relationships between them (by trace assertions). Note that the sets can be specified by sequences, the sequence "$\{a, b, c\}$" specifies the set consisting of the elements a, b, c. This convention is used for years and is easy to understand[3]. We need only an apparatus to make the states of individual objects distinct, to transform global states by both global calls (like *concatenate*, which affects more than one individual state, or *new*, which create a new local state), and local calls (like *PUSH*, which affects only one local state). The states of individual objects may be made distinct by adding individual labels to them. For instance $\{stack1 \mapsto PUSH(3).PUSH(5), stack2 \mapsto PUSH(3).PUSH(1).PUSH(8), stack3 \mapsto \varepsilon\}$ may represent a global state consisting of three stacks $stack1$, $stack2$, $stack3$, where the local state of $stack1$ is $PUSH(3).PUSH(4)$, the local state of $stack2$ is $PUSH(3).PUSH(1).PUSH(8)$ and $stack3$ is empty. The $stack1$, $stack2$ and $stack3$ are *unique labels* attached to appropriate canonical step-traces, creating *labeled step-traces*. This lead us to the concept of *uniquely labeled sets*.

15.1 Uniquely Labeled Sets

Let X be a set and $\mathcal{L}$ be a set of *labels*. A subset $\mathcal{X}$ of $\mathcal{L} \times X$ is a *labeled set*. We shall write $\alpha \mapsto x \in \mathcal{L} \times X$ instead of $(\alpha, x) \in \mathcal{L} \times X$. If $\mathcal{L} = \{1, 2, 3\}$, $X = \{a, b\}$ then $\{1 \mapsto a, 1 \mapsto b, 2 \mapsto a\}$ is an example of a labeled set.

A set $\mathcal{X} \subseteq \mathcal{L} \times X$ is *uniquely labeled* if for all $\alpha \in \mathcal{L}$ and for all $x, y \in X$,

$$(\alpha \mapsto x \in \mathcal{X} \ \wedge \ \alpha \mapsto y \in \mathcal{X}) \quad \Leftrightarrow \quad x = y.$$

[3] In a sense $\{a, b, c\}$ and $\langle a.b.c \rangle$ describe the same object, only the interpretation is different, see Section 7.

$\mathcal{X}$ is uniquely labeled if every element of it has an unambiguous label. For example $\{1 \mapsto a, 2 \mapsto a\}$ is uniquely labeled, while $\{1 \mapsto a, 1 \mapsto b, 2 \mapsto a\}$ is not. The *family of all uniquely labeled sets* over $\mathcal{L} \times X$ is denoted by $\mathcal{U}(\mathcal{L}, X)$. Note that $\emptyset$ is uniquely labeled, and for every $\mathcal{X} \in \mathcal{U}(\mathcal{L}, X)$, $|\mathcal{X}| \leq |\mathcal{L}|$. In particular we are interested in the family $\mathcal{U}(\mathcal{L}, \langle\Delta^*\rangle)$, where Δ is an alphabet.

For every uniquely labeled set $\mathcal{X} \subseteq \mathcal{L} \times X$, let $\mathcal{L}(\mathcal{X})$ be the set of all its labels,

$$\mathcal{L}(\mathcal{X}) = \{\alpha \in \mathcal{L} \mid \exists x \in X.\, \alpha \mapsto x \in \mathcal{X}\}.$$

For every $\alpha \in \mathcal{L}$ and every $\mathcal{X} \in \mathcal{U}(\mathcal{L}, X)$, let

$$\mathcal{X}|\alpha = x \quad \text{and} \quad \mathcal{X} \,\|\, \alpha = \alpha \mapsto x$$

if $\alpha \mapsto x \in \mathcal{X}$ for some x, and undefined otherwise. For instance if $\mathcal{X} = \{1 \mapsto a, 2 \mapsto b\}$ then $\mathcal{X}|1 = a$ and $\mathcal{X} \,\|\, 1 = 1 \mapsto a$, while $\mathcal{X}|3$ and $\mathcal{X} \,\|\, 3$ are undefined. The operator $|$ is called *"projection"* and $\|$ is called *"selection"*. Note that $\mathcal{X} \,\|\, \alpha = \alpha \mapsto \mathcal{X}|\alpha$, if $\mathcal{X}|\alpha$ is defined.

For every two uniquely labeled sets $\mathcal{X}$, $\mathcal{Y}$, we define the operation $\hookleftarrow$ and $\oplus$ as follows:

$$\begin{aligned} \mathcal{X} \hookleftarrow \mathcal{Y} &= \mathcal{X} \setminus \{\alpha \mapsto x \in \mathcal{X} \mid \alpha \in \mathcal{L}(\mathcal{Y})\} \cup \mathcal{Y}, \\ \mathcal{X} \oplus \mathcal{Y} &= \mathcal{X} \cup \mathcal{Y} \setminus \{\alpha \mapsto x \mid x \in X \wedge \alpha \in \mathcal{L}(\mathcal{X}) \cap \mathcal{L}(\mathcal{Y})\}. \end{aligned}$$

Note that $\mathcal{X} \hookleftarrow \mathcal{Y}$, $\mathcal{X} \oplus \mathcal{Y}$ are always uniquely labeled and $\mathcal{X} \oplus \mathcal{Y} = \mathcal{Y} \oplus \mathcal{X}$, but it may happen that $\mathcal{X} \hookleftarrow \mathcal{Y} \neq \mathcal{Y} \hookleftarrow \mathcal{X}$. The operation $\hookleftarrow$ replaces elements of $\mathcal{X}$ by the elements of $\mathcal{Y}$ with the same labels. If $\mathcal{L}(\mathcal{X}) \subseteq \mathcal{L}(\mathcal{Y})$ then $\mathcal{X} \hookleftarrow \mathcal{Y} = \mathcal{Y}$. If $\mathcal{L}(\mathcal{X}) \cap \mathcal{L}(\mathcal{Y}) = \emptyset$ then $\mathcal{X} \hookleftarrow \mathcal{Y} = \mathcal{X} \cup \mathcal{Y} = \mathcal{X} \oplus \mathcal{Y}$. For instance $\{1 \mapsto a, 2 \mapsto b\} \hookleftarrow \{1 \mapsto b\} = \{1 \mapsto b, 2 \mapsto b\}$, and $\{1 \mapsto a, 2 \mapsto b\} \oplus \{1 \mapsto b\} = \{2 \mapsto b\}$. The operator $\hookleftarrow$ is called the *labeled replacement*, the operator $\oplus$ is an auxiliary operator that is used to define concatenation and weak concatenation for the elements of $\mathcal{U}(\mathcal{L}, \langle\Delta^*\rangle)$.

The elements of $\mathcal{U}(\mathcal{L}, \langle\Delta^*\rangle)$ are called *uniquely labeled sets of step-sequences*, and the elements of $\mathcal{L} \times \langle\Delta^*\rangle$ *labeled step-sequences*. In particular $\emptyset$ and $\alpha \mapsto \varepsilon$ are labeled step-sequences. For every $\tau_1, \tau_2 \in \mathcal{U}(\mathcal{L}, \langle\Delta^*\rangle)$ we define *concatenation* "." and *weak concatenation* "$\smile$" by:

$$\begin{aligned} \tau_1.\tau_2 &= \{\alpha \mapsto t_1.t_2 \mid \alpha \mapsto t_1 \in \tau_1 \wedge \alpha \mapsto t_2 \in \tau_2\} \cup (\tau_1 \oplus \tau_2) \\ \tau_1 \smile \tau_2 &= \{\alpha \mapsto t_1 \smile t_2 \mid \alpha \mapsto t_1 \in \tau_1 \wedge \alpha \mapsto t_2 \in \tau_2\} \cup (\tau_1 \oplus \tau_2) \end{aligned}$$

Clearly $\tau_1.\tau_2$ and $\tau_1 \smile \tau_2$ are elements of $\mathcal{U}(\mathcal{L}, \langle\Delta^*\rangle)$. For instance if $\tau_1 = \{1 \mapsto \varepsilon, 2 \mapsto a.\langle b.a\rangle, 3 \mapsto a.a\}$ and $\tau_2 = \{1 \mapsto a.b, 2 \mapsto \langle c.d\rangle\}$, then we have $\tau_1.\tau_2 = \{1 \mapsto a.b, 2 \mapsto a.\langle b.a\rangle.\langle c.d\rangle, 3 \mapsto a.a\}$, $\tau_1 \smile \tau_2 = \{1 \mapsto a.b, 2 \mapsto a.\langle b.a.c.d\rangle, 3 \mapsto a.a\}$.

We also extend the operator $\in$, for any $\alpha \in \mathcal{L}$, $\tau \in \mathcal{U}(\mathcal{L}, \langle\Delta^*\rangle)$, $a \in \Delta$ by:

$$\begin{aligned} \alpha \in \tau &\iff \exists t \in \langle\Delta^*\rangle \,.\, \alpha \mapsto t \in \tau, \\ a \in \tau &\iff \exists \alpha \in \mathcal{L}.\, \exists t \in \langle\Delta^*\rangle \,.\, \alpha \mapsto t \in \tau \wedge a \in t. \end{aligned}$$

For instance, $2 \in \{1 \mapsto a.a, 2 \mapsto a.\langle b.a\rangle\}$, but $3 \notin \{1 \mapsto a.a, 2 \mapsto a.\langle b.a\rangle\}$, $b \in \{1 \mapsto a.a, 2 \mapsto a.\langle b.a\rangle\}$, but $c \notin \{1 \mapsto a.a, 2 \mapsto a.\langle b.a\rangle\}$.

15.2 Multi-Objects Trace Assertion Specifications

Let $TA = (\Delta_E, \mathcal{C}, \varrho, t_0)$ be a trace assertion specification, and let $\mathcal{L}$ be a set of labels. By a *free multi-object trace assertion specification* generated by TA and $\mathcal{L}$, we mean a tuple:

$$\mathcal{L}TA = (\mathcal{L} \times \Delta_E, \mathcal{U}(\mathcal{L}, \mathcal{C}), \varrho_{\mathcal{L}}, \emptyset)$$

where: $\mathcal{L} \times \Delta_E$ is the set of *labeled call-responses*, $\mathcal{U}(\mathcal{L}, \mathcal{C})$ is the *uniquely labeled set of canonical step-traces*, $\varrho_{\mathcal{L}} : \mathcal{L} \times \Delta_E \rightarrow \mathcal{U}(\mathcal{L}, \mathcal{C}) \leftrightarrow \mathcal{U}(\mathcal{L}, \mathcal{C})$ is the transition relation defined for all $\tau \in \mathcal{U}(\mathcal{L}, \mathcal{C})$ and $\alpha \mapsto a{:}d \in \mathcal{L} \times \Delta$ by:

$$\varrho_{\mathcal{L}}(\alpha \mapsto a{:}d)(\tau) = \begin{cases} \{\ \tau \hookleftarrow \{\alpha \mapsto t\} \ \mid \ t \in \varrho(a{:}d)(\tau|\alpha)\ \} & \text{if } \varrho(a{:}d)(\tau|\alpha) \neq \emptyset \\ \emptyset & \text{if } \varrho(a{:}d)(\tau|\alpha) = \emptyset \end{cases}$$

The above definition assumes *self-initialization* of modules, i.e. the first call initializes a given object in the module. Without self-initialization, the user must initialize an object before the call relating to this object. The pair $(\mathcal{L}, TA)$ describes $\mathcal{L}TA$ completely, since $\mathcal{L}TA$ is entirely specified by the specification TA and the description of $\mathcal{L}$. For instance $\mathcal{L}$ = *the set of all available names*, and TA from Figure 4 (without enhancement) describe completely the self-initializing multi-stack module. We may easily derive that in such a case (we specify normal behavior only so far): $\varrho_{\mathcal{L}}(st1 \mapsto PUSH(3))(\emptyset) = \{st1 \mapsto PUSH(3)\}$, while $\varrho_{\mathcal{L}}(st1 \mapsto POP)(\emptyset) = \emptyset$. If $\tau = \{st1 \mapsto PUSH(3).PUSH(1), st2 \mapsto \varepsilon, st3 \mapsto PUSH(5)\}$, then we have $\varrho_{\mathcal{L}}(st1 \mapsto POP)(\tau) = \{st1 \mapsto PUSH(3), st2 \mapsto \varepsilon, st3 \mapsto PUSH(5)\}$, while $\varrho_{\mathcal{L}}(st2 \mapsto POP)(\tau) = \emptyset$. In the sequel, except in the theory part, we shall prefer to write $PUSH(st1,3).PUSH(st1,1)$ instead of $st1 \mapsto PUSH(3).PUSH(1)$.

The normal behavior described by $\mathcal{L}TA$ is given by:

$$L(\mathcal{L}TA) = \{x \mid x \in (\mathcal{L} \times \Delta)^* \wedge \varrho^*_{\mathcal{L}}(x)(\emptyset) \neq \emptyset\},$$

where $\varrho^*_{\mathcal{L}}$ is the standard extension of $\varrho_{\mathcal{L}}$ onto $(\mathcal{L} \times \Delta)^* \rightarrow \mathcal{U}(\mathcal{L}, \mathcal{C}) \leftrightarrow \mathcal{U}(\mathcal{L}, \mathcal{C})$ (see Section 6.1).

The empty trace, ε, always does belong to $L(\mathcal{L}TA)$ since, by the definition, $\varrho^*_{\mathcal{L}}(\varepsilon)(\emptyset) = \{\emptyset\} \neq \emptyset$. For the self-initializing multi-stack trace assertion specification $\mathcal{L}TA$ we have $x = PUSH(st1,1).PUSH(st2,3).POP(st1) \in L(\mathcal{L}TA)$ since $\varrho^*_{\mathcal{L}}(x)(\emptyset) = \{st1 \mapsto \varepsilon, PUSH(st2,3)\} \neq \emptyset$, while $x.POP(st1) \notin L(\mathcal{L}TA)$ since $\varrho^*_{\mathcal{L}}(x.POP(st1))(\emptyset) = \emptyset$.

For a given $\mathcal{L}TA$, let $\varrho^{\emptyset}_{\mathcal{L}}$ denote the following transition relation, for all $\tau \in \mathcal{U}(\mathcal{L}, \mathcal{C})$ and $\alpha \mapsto a{:}d \in \mathcal{L} \times \Delta$:

$$\varrho^{\emptyset}_{\mathcal{L}}(\alpha \mapsto a{:}d)(\tau) = \begin{cases} \delta_{\mathcal{L}}(\alpha \mapsto a{:}d)(\tau) & \text{if } \alpha \in \tau \\ \emptyset & \text{if } \alpha \notin \tau \end{cases}$$

A *multi-object trace assertion specification* is a tuple:

$$MTA = (\mathcal{L}TA, \Delta_{E_{glob}}, \varrho_{glob})$$

where $\mathcal{L}TA$ as above, $\Delta_{E_{glob}}$ is the set of *global call-response events*, δ_{glob} the *global transition* relation, such that for some Σ_{glob}:

$$input(E_{glob}) = \bigcup_{i=0}^{\infty} (\Sigma_{glob} \times \mathcal{L}^i),$$
$$\varrho_{glob} \in \Delta_{glob} \rightarrow \mathcal{U}(\mathcal{L}, \mathcal{C}) \leftrightarrow \mathcal{U}(\mathcal{L}, \mathcal{C}).$$

We write $a(\alpha_1,\dots,\alpha_k) \in \Delta_{glob}$, and $a(\alpha_1,\dots,\alpha_k):d \in \Delta_{glob}$, rather than $(a,\alpha_1,\dots,\alpha_k)$, and $(a,\alpha_1,\dots,\alpha_k,d)$. For instance, we write $new(\alpha)$ instead of (new,α) and $concatenate(\alpha_1,\alpha_2,\alpha_3)$ instead of $(concatenate,\alpha_1,\alpha_2,\alpha_3)$.

For every *MTA* we define *the transition relation* $\hat{\varrho} \in ((\mathcal{L}\times\Delta)\cup\Delta_{glob}) \to \mathcal{U}(\mathcal{L},\mathcal{C}) \leftrightarrow \mathcal{U}(\mathcal{L},\mathcal{C})$, where for all $\tau \in \mathcal{U}(\mathcal{L},\mathcal{C})$ and $p \in (\mathcal{L}\times\Delta)\cup\Delta_{glob}$:

$$\hat{\varrho}(p)(\tau) = \begin{cases} \varrho_{\mathcal{L}}(p)(\tau) & \text{if } p \in \mathcal{L}\times\Delta \wedge new \notin \Sigma_{glob} \\ \varrho^{\emptyset}_{\mathcal{L}}(p)(\tau) & \text{if } p \in \mathcal{L}\times\Delta \wedge new \in \Sigma_{glob} \\ \varrho_{glob}(p)(\tau) & \text{if } p \in \Delta_{glob} \end{cases}$$

To create a new instance of an object, we use the access program call $new \in \Sigma_{glob}\times\mathcal{L}$, defined by:

$$\varrho_{glob}(new(\alpha))(\tau) = \begin{cases} \{\ \tau \hookleftarrow \{\alpha \mapsto t_0\}\ \} & \text{if } \alpha \in \tau \\ \emptyset & \text{if } \alpha \notin \tau \end{cases}$$

In other words $\hat{\varrho} = \varrho_{\mathcal{L}} \cup \varrho_{glob}$ if $new \notin \Sigma_{glob}$, and $\hat{\varrho} = \varrho^{\emptyset}_{\mathcal{L}} \cup \varrho_{glob}$ otherwise.

The global program $new(\alpha)$ is usually accompanied by a program $delete(\alpha) \in \Sigma_{glob}\times\mathcal{L}$, which deletes the instance of *TA* with label α:

$$\varrho_{glob}(delete(\alpha))(\tau) = \begin{cases} \{\ \tau \setminus \{\tau \| \alpha\}\ \} & \text{if } \alpha \in \tau \\ \emptyset & \text{if } \alpha \notin \tau \end{cases}$$

The *normal behavior* generated by *MTA* is defined by:

$$L(MTA) = \{x \mid x \in ((\mathcal{L}\times\Delta)\cup\Delta_{glob})^* \wedge \hat{\varrho}^*(x)(\emptyset) \neq \emptyset\}.$$

For instance ε always does belong to $L(MTA)$ since $\hat{\varrho}^*(\varepsilon)(\emptyset) = \{\emptyset\} \neq \emptyset$, if $new, delete \in \Sigma_{glob}$, then $new(\alpha_1).new(\alpha_2).delete(\alpha_2) \in L(MTA)$ since $\hat{\varrho}^*(new(\alpha_1).new(\alpha_2).delete(\alpha_2))(\emptyset) = \{\alpha_1 \mapsto t_0\} \neq \emptyset$, and $\hat{\varrho}^*(new(\alpha_1).new(\alpha_2).delete(\alpha_2).delete(\alpha_1))(\emptyset) = \{\emptyset\} \neq \emptyset$, while $new(\alpha_1).delete(\alpha_2) \notin L(MTA)$ since $\hat{\varrho}^*(new(\alpha_1).delete(\alpha_2))(\emptyset) = \emptyset$.

MTA is *self-initialized* if $new \notin \Sigma_{glob}$, *output independent* if *TA* is output independent, *deterministic* if *TA* is deterministic and $|\delta_{glob}(\tau,p)| \leq 1$. The concepts of *enhancement, full specification, Mealy form* and *state constructors* can easily be introduced for multi-objects modules. The counterparts of Lemma 10.1, Proposition 10.2 and Proposition 12.1 also do hold for multi-objects trace specifications. Figure 9 represents a full self-initialized multi-object trace assertion specification for the Cross module that was introduced and analyzed in [14]. The specification of the Cross module caused some problems when the older convention and techniques were used [14]. The Cross specification is non-deterministic and output independent. The module implements up to two sets, labeled by either *a* or *b*, each set may contain 0, 1, both 0 and 1 or is empty. There are two local operations *INSERT*, which inserts an element into a given set, *TEST* which tests if an element is in a given set, and one global operation *CROSS* which takes two sets and divides non-deterministically their union into two disjoint sets. The module is self-initializing, the first *INSERT* creates a set. Figure 10 represents the full multi-objects trace assertion specification of Multi-Stack with *new, concatenate,* and *delete* as global program calls. The specification is deterministic, output-independent and non self-initializing.

Labels
$\mathcal{L} = \{a, b\}$

Syntax of Access Programs

Name	Type	Argument	Value	Call-Response Forms	Code
INSERT	local	0 or 1		$\mathit{INSERT}(*, i){:}\mathit{nil}$	i
TEST	local	0 or 1	*Boolean*	$\mathit{TEST}(*, i){:}d$	
CROSS	global			$\mathit{CROSS}{:}\mathit{nil}$	

Local Canonical Step-traces (Coded)
$\mathit{canonical}(t) \quad \Leftrightarrow \quad (t = \varepsilon \vee t = i \vee t = \langle i_1.i_2\rangle) \wedge i, i_1, i_2 \in \{0,1\} \wedge i_1 \neq i_2$
$t_0 = \varepsilon$

Local Trace Assertions
$\varrho(\mathit{INSERT}(*, i))(t) =$

Equivalence
$t \smile i$

$\varrho(\mathit{TEST}(*, i){:}d)(t) =$

Condition	Equivalence
$i \in t \wedge d = \mathit{true}$	t
$i \notin t \wedge d = \mathit{false}$	t

Global Canonical Step-traces (Redundant)

$$\begin{aligned} \mathit{global\_canonical}(\tau) \Leftrightarrow\ & \tau = \emptyset \vee ((\tau = \{a \mapsto t\} \vee \tau = \{b \mapsto t\}) \wedge \mathit{canonical}(t)) \\ & \vee(\tau = \{a \mapsto t_1, b \mapsto t_2\} \wedge \mathit{canonical}(t_1) \wedge \mathit{canonical}(t_2)) \\ \tau_0 = \emptyset & \end{aligned}$$

Global Trace Assertions
$\hat{\varrho}(\mathit{CROSS})(\tau) =$

Condition	Clusters			
$0 \in \tau \wedge 1 \in \tau \wedge \lvert\tau\rvert = 2$	$\{a \mapsto \langle 0.1\rangle, b \mapsto \varepsilon\}$	$\{a \mapsto 0, b \mapsto 1\}$	$\{a \mapsto 1, b \mapsto 0\}$	$\{a \mapsto \varepsilon, b \mapsto \langle 0.1\rangle\}$
$0 \in \tau \wedge 1 \notin \tau \wedge \lvert\tau\rvert = 2$	$\{a \mapsto \varepsilon, b \mapsto 0\}$		$\{a \mapsto 0, b \mapsto \varepsilon\}$	
$1 \in \tau \wedge 0 \notin \tau \wedge \lvert\tau\rvert = 2$	$\{a \mapsto \varepsilon, b \mapsto 1\}$		$\{a \mapsto 1, b \mapsto \varepsilon\}$	
$\tau = \{a \mapsto \varepsilon, b \mapsto \varepsilon\}$	τ			
% $\lvert\tau\rvert < 2$	τ			

Extended Local Trace Assertions (Redundant, except for enhancements)
$\hat{\varrho}(\mathit{INSERT}(\alpha, i))(\tau) =$

Equivalence
$\tau \smile \{\alpha \mapsto i\}$

$\hat{\varrho}(\mathit{TEST}(\alpha, i){:}d)(\tau) =$

Condition	Equivalence
$\alpha \mapsto i \in \tau \wedge d = \mathit{true}$	τ
$\alpha \in \tau \wedge \alpha \mapsto i \notin \tau \wedge d = \mathit{false}$	τ
% $\alpha \notin \tau$	τ

Figure 9: Enhanced Trace Assertion Specification for (a self-initializing) Cross Module

Labels $\mathcal{L}$ =*available names*

Syntax of Access Programs

Name	Type	Argument	Value	Call-Response Forms	Codes
POP	local			$POP(*){:}nil$	
PUSH	local	*integer*		$PUSH(*, i){:}nil$	i
TOP	local		*integer*	$TOP(*){:}i$	
new	global	*label*		$new(*)$	
concatenate	global	$3 \times$ *label*		$concatenate(*, *, *)$	
delete	global	*label*		$delete(*)$	

Local Canonical Step-traces

$canonical(t) \quad \Leftrightarrow \quad t = [d_i]_{i=1}^{n} \wedge 0 \leq n \leq size$

$t_0 = \varepsilon$

Local Trace Assertions

$\varrho(POP(*))(t) =$

Trace Patterns	Equivalence
$t = s.d$	s
% $t = \varepsilon$	ε

$\varrho(PUSH(*, d))(t) =$

Condition	Equivalence
$length(t) < size$	$t.d$
% $length(t) = size$	t

$\varrho(TOP(*){:}d)(t) =$

Condition	Trace Patterns	Equivalence
	$t = s.d$	t
% $d = nil$	$t = \varepsilon$	ε

Global Canonical Traces (Redundant)

$global\_canonical(\tau) \quad \Leftrightarrow \quad \tau = \{\alpha_i \mapsto t_i\}_{i=1}^{k} \wedge \bigwedge_{i=1}^{t_i} canonical(t_i) \wedge (\alpha_j = \alpha_l \Leftrightarrow j = l)$

$\tau_0 = \emptyset$

Global Trace Assertions

$\hat{\varrho}(new(\alpha))(\tau) =$

Condition	Equivalence
$\alpha \notin \tau$	$\tau \hookleftarrow \{\alpha \mapsto \varepsilon\}$
% $\alpha \in \tau$	τ

$\hat{\varrho}(concatenate(\alpha_1, \alpha_2, \alpha_3)(\tau) =$

Condition	Equivalence
$\alpha_1 \in \tau \wedge \alpha_2 \in \tau \wedge \alpha_3 \in \tau$	$\tau \hookleftarrow \{\alpha_3 \mapsto \tau\|\alpha_1.\tau\|\alpha_2\}$
% $\alpha_1 \notin \tau \vee \alpha_2 \notin \tau \wedge \alpha_3 \notin \tau$	τ

$\hat{\varrho}(delete(\alpha))(\tau) =$

Condition	Equivalence
$\alpha \in \tau$	$\tau \setminus \{\tau \\| \alpha\}$
% $\alpha \notin \tau$	τ

Dictionary

size : the size of the stack, $length(t)$: the length of the trace t

Figure 10: Enhanced Trace Assertion Specification for Multiple Stack Module. Extended Local Trace Assertions are omitted as redundant.

16 Trace Assertion Method and Algebraic Specification

There are strong similarities between the trace assertion method and the *algebraic specification method* (see [7, 34]) for specifying abstract data types. Examples of similarities are:

1. Syntax parts of trace assertion specifications correspond to signatures in algebraic specification.
2. For output-independent trace assertion specifications, trace assertions correspond to conditional equations,
3. Canonical traces corresponds canonical terms (see [7]).
4. State constructors (as introduced in the paper to solve the problem that it is not always possible to represent the possible states uniquely by sequences of call-response pairs of visible functions) correspond to auxiliary/hidden functions in algebraic specification.

However, there are major differences. The **main difference** is that

1. *Algebraic specification supports* **implicit equations** while trace assertion method uses **explicit equations** only.

The functions *PUSH*, *POP*, and *TOP* operating on non-empty stack may abstractly be *implicitly* defined as [7]:

$$POP(PUSH(s,a)) = s$$
$$TOP(PUSH(s,a)) = a$$

Less abstract, with states of the stack represented as sequences and "." denoting concatenation, *explicit* definition of the same part of stack is the following (also see [7]):

$$\begin{aligned} PUSH(s,a) &= s.a \\ POP(s.a) &= s \\ TOP(s.a) &= a \end{aligned}$$

In the second, explicit, case we may replace "$=$" by "$\overset{df}{=}$", but in the first, implicit, case we cannot. The trace assertion specification is a straight abstraction of the second case. The *implicit* definitions might sometimes be shorter, they are usually more abstract. However, typically *explicit* definitions are considered to be more readable and easier to understand. The stack is well-known and easy to understand module, but even here some students have encountered initial problems to understand that the implicit equations really define the stack, while the explicit equations are practically self-explaining. For more complex modules, as for example parts of protocols [5, 12], parts of software for aircraft control [31], or intra-processor, inter-process communication via mailboxes [32] both defining and understanding implicit equations might be difficult (how simple would equational definitions of the Unique Integer or the Cross module look like?).

The second difference is

2. The underlying models for algebraic specification are *abstract algebras* [4], while the underlying model for trace assertion method are *automata*.

While as we mentioned before there is similarity between automata and algebras, they are different models. The other differences:

3. To specify in trace assertion specifications that a function does not change the state, it is necessary to explicitly write trace assertions expressing this, while in algebraic specification it is possible already in the signature to express this so that there is no need for equations.

4. Trace assertion specifications provide syntactic facilities which makes it possible in certain cases to specify a function by a single trace assertion, where the use of *auxiliary/hidden functions* (e.g. in the definition of a stack with overflow [3]) or *recursive definitions* (e.g. in the definition of the dequeue function for a queue [7]) are necessary in algebraic specifications.

5. *State constructors* that correspond to *auxiliary/hidden* functions are used only to handle heavy non-determinism, while the use of auxiliary/hidden functions is much wider in algebraic specifications.

Suppose for instance that *PUSH* additionally returns the value that is pushed on the top of the stack. The trace assertion specification requires only small adjustments, in Figure 5 we need to replace $PUSH(d){:}nil$ by $PUSH(d) : d$ in the last column of the Syntax of Access Programs, $t = [PUSH(d_i)]_{i=1}^{n}$ by $t = [PUSH(d_i){:}d_i]_{i=1}^{n}$ in the Canonical Step-traces definition (or *nothing* if codes are used as in Figure 11), and

$\varrho(PUSH(d))(t) =$

Condition	Equivalence
$length(t) < size$	$t.PUSH(d)$
% $length(t) = size$	t

by:

$\varrho(PUSH(d){:}d)(t) =$

Condition	Equivalence
$length(t) < size$	$t.PUSH(d){:}d$
% $length(t) = size$	t

Similar minor adjustments are needed for the Mealy form of Figure 6. The algebraic specification requires the use of an *auxiliary/hidden* function *push*, and may look like:

$$\begin{aligned} POP(push(s,a)) &= s \\ TOP(push(s,a)) &= a \\ PUSH(s,a) &= (push(s,a),a) \end{aligned}$$

where $PUSH : Stack \times integer \rightarrow Stack \times integer$. The descriptive power of trace assertion specification and algebraic specification is the same. Every trace assertion specification can be transformed into an equivalent *canonical terms algebra* ([7, 34]), and for every algebraic specification, a trace assertion specification equivalent to the *canonical terms algebra* of the given algebraic specification can be constructed. The constructions in the general case are formally complex and tedious, even so the intuitions seem to be clear. We will show how such transformations may look like in some special cases. Those transformations will also emphasize similarities and differences.

A trace assertion specification $TA = (\Delta_E, \mathcal{C}, \varrho, t_0)$ is *total* if it defines a transition for all calls, i.e. $\kappa(t,a) = 1$ for all $t \in \mathcal{C}$ and $a \in \Sigma_E$.

Let $TA = (\Delta_E, \mathcal{C}, \varrho, t_0)$ be a *deterministic and total* trace assertion specification. We define the many-sorted algebra

$$\mathcal{A}_{TA} = (Sort_{\mathcal{C}}, Sort_{\mathcal{O}}, Sort_1, \ldots, Sort_{k_{TA}}; Op)$$

where $Sort_{\mathcal{C}} = \mathcal{C}$, $Sort_{\mathcal{O}} = \mathcal{O}_E$, $Sort_1, \ldots, Sort_{k_{TA}}$ are the domains of the arguments of the procedures (function calls) from $\mathcal{N}_\Sigma$, and

$$Op = \{\hat{f} \mid f \in E\} \cup \{\tilde{f} \mid f \in E\},$$

where $\hat{f}$ and $\tilde{f}$ are the functions defined as follows:

$$\varrho(f(d_1, ..., d_r) : d)(t) = \{s\} \iff \begin{cases} \hat{f}(t, d_1, ..., d_r) = s \\ \tilde{f}(t, d_1, ..., d_r) = d \end{cases}$$

For *TA* representing the Stack module (Figure 4) the above transformation result in the following two-sorted function algebra

$$\mathcal{A}_{TA} = (Sort_{\mathcal{C}}, Sort_{\mathcal{O}}, Sort_1; Op)$$

with $Sort_{\mathcal{C}} = \mathcal{C} = \{[PUSH(d_i)]_{i=1}^{n} \mid 0 \leq n \leq size \cap d_i \in integer\}$, $Sort_{\mathcal{O}} = \mathcal{O}_E = integer \cup \{nil\}$, $Sort_1 = integer$, $Op = \{\widehat{PUSH}, \widetilde{PUSH}, \widehat{POP}, \widetilde{POP}, \widehat{TOP}, \widetilde{TOP}\}$, $\widehat{PUSH} : Sort_{\mathcal{C}} \times Sort_1 \to Sort_{\mathcal{C}}$, $\widetilde{PUSH} : Sort_{\mathcal{C}} \times Sort_1 \to Sort_{\mathcal{O}}$, $\widehat{POP} : Sort_{\mathcal{C}} \to Sort_{\mathcal{C}}$, $\widetilde{POP} : Sort_{\mathcal{C}} \to Sort_{\mathcal{O}}$, $\widehat{TOP} : Sort_{\mathcal{C}} \to Sort_{\mathcal{C}}$, $\widetilde{TOP} : Sort_{\mathcal{C}} \to Sort_{\mathcal{O}}$, and for every $t \in \mathcal{C}$, and every integer i:

$$\begin{array}{lll}
\widehat{PUSH}(t, i) & = t.PUSH(i) & \text{if } lenght(t) < size, \text{ and,} \\
\widehat{PUSH}(t, i) & = t & \text{if } lenght(t) = size, \\
\widetilde{PUSH}(t, i) & = nil & \\
\widehat{POP}(t.PUSH(i)) & = t & \text{and } \widehat{POP}(\epsilon) = \epsilon, \\
\widetilde{POP}(t) & = nil & \\
\widehat{TOP}(t) & = t & \\
\widetilde{TOP}(t.PUSH(i)) & = i & \text{and } \widetilde{TOP}(\epsilon) = nil,
\end{array}$$

We say that the "tilde" function $\tilde{f}$ is *trivial* if $range(\tilde{f}) = \{nil\}$, and that the "hat" function $\hat{f}$ is *trivial* if $\hat{f}(t, d_1, \ldots, d_r) = t$ for every t. For the Stack example, the functions $\widetilde{PUSH}$, $\widetilde{POP}$ and $\widehat{TOP}$ are trivial.

Let $\mathcal{A}_{TA}^{modified}$ be the algebra derived from $\mathcal{A}_{TA}$ by eliminating all trivial functions. We will consider $\mathcal{A}_{TA}^{modified}$ as an algebraic equivalent of the deterministic trace assertion specification *TA*.

For non-deterministic trace assertion specifications we proceed in a similar manner, but instead of standard many-sorted algebras we have to use for instance *partial algebras* (see [34] chapter 3.3.5).

Consider a many-sorted algebra $\mathcal{A} = (S_0, S_1, \ldots, S_k; Op)$. We say that a sort S_i is *domestic* if for every $f \in Op$, S_i is a component of the domain of f, and there exists at least one $g \in Op$ such that S_i is the range of g. The sort is called *foreign* if it is not domestic. For instance for the two sorted algebra that defines stack of integers, the sort *stack* is domestic and the sort *integers* is foreign. Intuitively, the domestic sort is defined by the algebra, and all foreign sorts are predefined by other means.

An element of $s_o \in S_0$ is called a *generator* of S_0 (see [34]), if every element of S_0 can be derived from s_0 by applying a sequence of operators (functions) from *Op*.

To transform an algebraic specification in a trace assertion specification we have to resolve following main problems:

1. all *implicit* equations must be replaced by *explicit* ones,

2. only one domestic sort is allowed,

3. there exists a generator s_0 of S_0.

It appears that many algebraic specifications can be transformed into the form described above, however the result is usually less general.

Let $\mathcal{A} = (S_0, S_1, \ldots, S_k; Op)$ be a many-sorted algebra with explicit equations, one domestic domain S_0, and suppose s_0 is a generator of S_0. Without loss of generality we may assume that for every $f \in Op$, the domain of f is of the form $S_0 \times S_{i_1} \times \ldots \times S_{i_f}$, i.e. the value of the first argument of f belongs to S_0, and that S_0 is the set of *canonical terms* [7, 34] (so in the case of stack, instead of $\langle 1, 2\rangle \in S_0$, we have $PUSH(PUSH(stack, 1), 2) \in S_0$).

Define E, the set of access program names as $E = \{\mathbf{f} \mid f \in Op\}$. For all $\mathbf{f} \in E$, we defined $input(\mathbf{f})$ and $output(\mathbf{f})$ to be the smallest sets such that:

$$\begin{array}{lcl} (v_1, ..., v_k) \in input(\mathbf{f}) \wedge nil \in output(\mathbf{f}) & \iff & \exists t, s \in S_0 \, . \, f(t, v_1, ..., v_r) = s \\ (v_1, ..., v_k) \in input(\mathbf{f}) \wedge d \in output(\mathbf{f}) & \iff & \exists t \in S_0 \, . \, f(t, v_1, ..., v_k) = d. \end{array}$$

Let Θ be the following mapping that transforms S_0 into a set of traces over Δ_E: $\Theta(s_0) = \epsilon$ and for every $f(v_0, v_1, \ldots, v_k) \in S_0$, $\Theta(f(v_0, v_1, \ldots, v_k) = \Theta(v_0).\mathbf{f}(v_1, \ldots, v_k){:}nil$. Now define

$$TA = (\Delta_E, \Theta(S_0), \varrho, \epsilon)$$

where:

$$\begin{array}{lcl} \varrho(\mathbf{f}(v_1, \ldots, v_k){:}nil)(s) = \{t\} & \iff & f(s, v_1, \ldots, v_k) = t \\ \varrho(\mathbf{f}(v_1, \ldots, v_k){:}d)(s) = \{s\} & \iff & f(s, v_1, \ldots, v_k) = d. \end{array}$$

and $\varrho(a{:}d)(s) = \emptyset$ for all other cases. We shall consider TA as a trace assertion specification that is equivalent to the algebra $\mathcal{A}$.

We believe the areas of applications for the algebraic specifications are different than for the trace assertion method. The algebraic specification is better suited for defining abstract data types in programming languages (as SML, LARCH, etc., see [34]). The trace assertion method is better suited for specifying complex interface modules as for instance communication protocols [5, 12, 31, 32]. The division follows from the general pattern of applicability of automata based and algebraic models. One may model integers as an automaton (it is usually defined as an algebra), or may define the semantics of SCR specification [11] as an abstract algebra (it is defined as a kind of automaton), however in both cases the advantage as such way of modeling is hardly seen.

17 Final Comment

An automata-based model for the trace assertion method has been presented and its formal consistency has been proven. A modified specification format based on this model has also been proposed. The main points of the model are the following:

- the alphabet which represent observable event occurrences is built from call-response events,

- the structure of the trace assertion specification is entirely described on the bases of normal behavior only,
- the refinement relation captures the externally observable behavior of module specifications,
- trace assertion specifications can be refined into "more deterministic" and "more total" trace assertion specifications or into module specifications with some "more concrete" state space, using a simulation relation,
- exceptional behavior is specified separately as an enhancement of normal behavior, and such an enhancement may be added to the trace assertion specification, leading a behavioral refinement,
- canonical step-traces (instead of canonical traces) are used to specify states for single-object modules, and sets of canonical step-traces are used to specify states for multi-object modules. Sequence notation is used to specify both step-traces and sets of step traces,
- Mealy forms are special cases of a more general yet simpler model,
- multi-object modules are specified using the concept of uniquely labeled sets of step-traces.

Neither the monitored events [11, 27, 33] nor non-sequential modules are considered in this paper. For non-sequential models a possible delay between a call and its response must be modeled, so "true-concurrency" models should rather be used [18]. We have shown that the output value functions are redundant. The theory does not need them, and we believe they usually make specifications less readable. We have found the standard forms shorter and more readable than the Mealy forms. For the output dependent trace specifications, the explicit output functions seem to be useless at all. The specification of multi-object modules is not much different than single-object modules. The trace assertion method and algebraic specification can be seen as complimentary approaches. They have some things in common, but substantial differences as well. The main difference is the use of implicit equations in algebraic specifications, and explicit equations only in trace assertions. Their areas of applications seem to be different.

Acknowledgment

We would like to thank D. Parnas and A. J. van Schouwen for many useful comments and discussions. We gratefully acknowledge three anonymous referees, whose comments significantly contributed to the final version of this paper.

References

[1] J.-R. Abrial. *The B Book: Assigning Programs to Meaning.* Cambridge University Press, 1996.

[2] A. Arnold. *Finite Transition Systems.* Prentice Hall, 1994.

[3] W. Bartussek and D.L. Parnas. Using assertions about traces to write abstract specifications for software modules. In G. Bracchi and P. C. Lockemann, editors, *2nd Conf. on European Cooperation in Informatics on Information Systems Methodology*, Lecture Notes in Computer Science 65, pages 211–236, Venice, Italy, 1978. Springer-Verlag.

[4] P. M. Cohen. *Universal Algebra*. D. Reidel, 1981.

[5] F. Courtois and D. L. Parnas. Formally specifying a communication protocol using the trace assertion method. Technical Report CRL Report No. 269, McMaster University, Hamilton, Ontario, Canada, 1993.

[6] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, Singapore, 1995.

[7] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs in Theoretical Computer Science. Springer-Verlag, Berlin ; New York, 1985.

[8] S. Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, New York, 1974.

[9] R. Fräisse. *Theory of Relations*. North Holland, 1986.

[10] Jifeng He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *European Symposium on Programming*, Lecture Notes in Computer Science 213. Springer-Verlag, 1986.

[11] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specification. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, 1996.

[12] D. M. Hoffman. The trace specification of communication protocols. *IEEE Transactions on Computers*, 34(12):1102–1113, 1985.

[13] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computations*. Addison-Wesley, Reading, MA, 1979.

[14] M. Iglewski, M. Kubica, and J. Madey. Trace specification of non-deterministic multi-object modules. Technical Report 95-05(205), Institute of Informatics, Warsaw University, Warsaw, Poland, 1995.

[15] M. Iglewski, J. Madey, and K. Stencel. On fundamentals of the trace assertion method. Technical Report RR 94/09-6, Université du Québec à Hull, Hull, Canada, 1994.

[16] M. Iglewski, J. Mincer-Daszkiewicz, and J. Stencel. Some experiences with specification of non-deterministic modules. Technical Report RR 94/09-7, Université du Québec à Hull, Hull, Canada, 1994.

[17] R. Janicki. Towards a formal semantics of parnas tables. In *17th International Conference on Software Engineering*, pages 231–240, Seattle, Washington, USA, 1995. ACM Press.

[18] R. Janicki and M. Koutny. Structure of concurrency. *Theoretical Computer Science*, 112(1):5–52, 1993.

[19] R. Janicki, D. L. Parnas, and J. Zucker. Tabular representations in relational documents. In C. Brink and G. Schmidt, editors, *Relational Methods in Computer Science*. Springer-Verlag, 1997.

[20] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[21] N. Lynch and F. Vaandrager. Forward and backward simulations: I. untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[22] J. McLean. A formal foundations for the abstract specification of software. *Journal of the ACM*, 31(3):600–627, 1984.

[23] H. D. Mills. Stepwise refinement and verification in box-structure systems. *Computer*, pages 23–26, 1988.

[24] T. Norvell. On trace specifications. CRL Report 305, McMaster University, Hamilton, Canada, 1995.

[25] D. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, 1972.

[26] D. Parnas, J. Madey, and M. Iglewski. Precise documentation of well-structured programs. *IEEE Transactions on Software Engineering*, 20(12):948–976, 1994.

[27] D. Parnas and Y. Wang. The trace assertion method of module interface specification. Technical Report 89–261, CIS, Queen's University, 1989.

[28] S. J. Prowell. *Sequence-Based Software Specification*. Ph.D. Thesis, University of Tennessee, 1996.

[29] G. Rozenberg and R. Varraedt. Subset languages of Petri nets. *Theoretical Computer Science*, 26:301–323, 1983.

[30] G. Schmidt and T. Ströhlein. *Relations and Graphs: Discrete Mathematics for Computer Scientists*. EATCS Monographs in Theoretical Computer Science. Springer-Verlag, Berlin ; New York, 1993.

[31] A. J. van Schouwen. The A-7 requirements model: Re-examination for real time systems and an application to monitoring systems. Technical Report 90–276, Queen's University, 1990.

[32] A. J. van Schouwen. On the road to practical module interface specification. In *Lecture presented at McMaster Workshop on Tools for Tabular Notations*, McMaster University, Hamilton, Ontario, Canada, 1996.

[33] Y. Wang. *Specifying and Simulating the Externally Observable Behaviour of Modules*. Ph.D. Thesis, McMaster University, 1994. also CRL Report 292, TRIO.

[34] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 2, pages 675–788. Elsevier Science, 1990.

Biographies

Ryszard Janicki is a Professor at the Department of Computing and Software, McMaster University, Hamilton, Canada. He received the M.Sc. degree in Applied Mathematics from the Warsaw University of Technology, Poland in 1975, and the Ph.D. and Habilitation in Computer Science from the Polish Academy of Sciences, Warsaw, Poland in 1977 and 1981 respectively. He taught computer science and mathematics at the Warsaw University of Technology, Poland in 1975-1984, Aalborg University, Denmark in 1984-86, before joining McMaster in 1986. He was a Visiting Scholar at University of Newcastle upon Tyne, U.K., in 1982 and a Visiting Professor at Bordeaux University, France, in 1994-95.

He published more than 80 papers and co-authored a monograph. His research interests include concurrency theory, fundamentals of software engineering, ranking theory, and relational methods in computer science.

Emil Sekerinski is Assistant Professor at the Department of Computing and Software, McMaster University, Hamilton, Canada. He studied computer science in Stuttgart and Karlsruhe, Germany, and received the diploma in computer science and the doctoral degree from the University of Karlsruhe, in 1989 and 1994, respectively. He had positions at Åbo Akademi, Turku, Finland, 1995-1997 before joining McMaster in 1997.

He published more than 20 papers and co-edited a book. His research interests include mathematical specification and development techniques, object orientation, concurrent and reactive systems, programming languages and programming tools.