Foundations of the Trace Assertion Method of Module Interface Specification

Ryszard Janicki *
Department of Computer Science and Systems
McMaster University
Hamilton, Ontario, Canada L8S 4K1
janicki@mcmaster.ca

Abstract

The trace assertion method is a formal state machine based method for specifying module interfaces ([3, 15, 25, 28, 32, 36]). 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. A formal model for the trace assertion method is proposed. The concept of *step-traces* is introduced and applied. The role of non-determinism, normal and exceptional behaviour, value functions and multi-object modules are discussed. The relationship with the Algebraic Specification ([9, 37]) is analyzed.

Contents

1 Introduction **2**

2 Introductory Examples **4**

3 Alphabet **6**

4 Normal and Exceptional Behaviour **7**

5 Value Functions **8**

6 Languages and Automata **9**
6.1 Deterministic and Non-deterministic Automata 9
6.2 Mealy Machines vs Automata 10
6.3 Right Congruences 11
6.4 Consequences for Trace Assertion Method 13

7 Defining Objects by Sequences **14**

*Supported by NSERC Research Grant.

8 Trace Only Automata **17**

9 Trace Assertion Specifications **19**

10 Mealy Forms of Trace Assertion specification **21**

11 Trace Assertion Specifications with State Constructors **23**

12 Enhancements and Full Trace Assertion Specifications **24**

13 Specification Format **26**

14 Multi-Object Modules **28**
14.1 Uniquely Labeled Sets 31
14.2 Multi-Objects Trace Assertion Specifications 34
14.3 Format for Multi-Objects Trace Assertion Specification 37

15 Trace Assertion Method and Algebraic Specification **37**

16 Final Comment **43**

1 Introduction

Software modules, viewed as "black boxes" [28, 26], hide some software decisions and provide abstract data types. They could conveniently be specified using the *trace assertion method*. A trace is a complete history of the visible behaviour of the objects. It includes all events affecting the object, eventually with the outputs produced. Formally a trace is a sequence of event occurrences. The fundamental principle is that a trace specification describes only those features of an object that are externally observable and the central idea of the approach is that the traces can be divided into clusters and each cluster is represented by a single canonical trace. Frequently these clusters are equivalence classes, and then the module specification is called deterministic. A simple, automata based, model for the traces assertions presented below. The problem of non-determinism and output values is emphasized.

The trace assertion method was first formulated by Bartussek and Parnas in [3], as a possible answer for some problems with early algebraic specifications [9, 37], and since then has undergone many modifications [15, 25, 32, 36]. Many persons have been involved in the development of the Trace Assertion Method, but the main initial ideas are due to D. L. Parnas. In recent years, there has been an increased interest in the Trace Assertion Method [17, 18, 19, 27, 31, 35], however fully satisfactory foundations have not yet been developed.

Like many others currently used techniques (object oriented programming, algebraic specification, etc.), the sequence-based methods for software analysis and specification have been born in seventies. Initially they were mainly used to analyze and to prove various properties of programming schemes ([4, 12, 20, 24]). Among others they were used to prove that, under the same set of operations, recursive programs have greater computational power than the iterative programs, and the programs with coroutines are more powerful than the recursive ones.

Later the sequence-based techniques were used for the specification purposes ([3, 14]). The first stream has reached a saturation point in late seventies, the second is very much alive.

The term "trace" seems to be overused in Computer Science. It has at least two different meanings. First, the primary meaning in North America, is just a sequence of events, actions, operations, or systems calls, i.e. it is a sequence of specially interpreted elements. The second, primary in Europe, is an element of a partially commutative monoid, where the monoid operation is a concatenation (see [8]). In the second case the name "Mazurkiewicz traces" is often used [8, 22] . 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 trace assertion method is a vital part of general, relation based tabular specification technique, which had several serious industrial applications (see [23, 30]).

The trace assertion method is based on the following postulates:

- *Information hiding (Black box)* principle [28, 26] is fundamental for any specification.
- *Sequences* are natural, powerful and easy to use tools for specifying abstract *objects*.
- *Explicit equations* are preferable over *implicit equations*. Implicit equations might provide shorter and more abstract specification, but are much less readable and difficult to derive that the explicit ones (see Section 15).
- *State machines* are natural, powerful and easy to use tools for specifying *systems*. For many applications they are better than algebras, and their use for specification is growing [1, 2, 13]. 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:

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

  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 [10, 5] and Section 15.

The four next sections are quite informal. In the next section we will introduce and briefly discuss three simple modules. These modules will be used to illustrate all the major problems and solutions. In Section 3 the question "What is an atomic observable event?" is discussed. The question "What should be done with possible misuses (as for instance popping the empty stack) of a module?" is analyzed in Section 4. 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 5. More formal part starts with Section 6, where a collection of rather well known results from the formal language and automata theory that are used in or influenced the trace assertion method, is presented. Objects described by the sequences and the concept of step-sequences is 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, which is a automata-like structure is given in Section 9. The special instances of the Trace Assertion Specification as Mealy form and the controversial use of invisible actions

are discussed respectively in Sections 10 and 11. One of the results obtained is that we do much better without output value functions. The theory is simpler and more consistent, no information is lost, and the resulting specification seems to be more readable. The misuses are formally discussed in Section 12. The idea is that the misuses are modeled separately and eventually they may be added to the pure trace specification as an enhancement. The result is called a full trace specification. Section 13 defines a format for the trace specification technique. All the examples from Section 2 are formally specified in this format at the end of this section. Section 14 deals with multi-object modules. The uniquely labeled sets of step-traces are introduced and used as a specification tool. The relationship between the Trace Assertions and Algebraic Specifications ([9, 37]) is analyzed in Section 15. 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:

- $PUSH(i)$, which enters an integer i on the stack,
- POP, which has no arguments and removes the top of the stack, and
- TOP, which has 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 $PUSH(4).PUSH(1).POP.PUSH(7).TOP$ and $PUSH(4).PUSH(7)$ both lead to state $\langle 4, 7\rangle$. They could be seen as equivalent and we can choose for instance the trace $PUSH(4).PUSH(7)$ as a *canonical* trace representing the state $\langle 4, 7\rangle$.

The Unique Integer module provides only one access program:

- GET, which 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 has been returned by previous GET invocations. Intuitively the state of Unique Integer module is determined by the set of all integers that have been returned by all previous GET invocations. In this case the sequence, say $GET.GET.GET$, 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 an action-response event, with the action GET and the response i. Any trace built from $GET{:}i$'s 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 a 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. $\langle\cdot\rangle$ is an operator that makes order irrelevant, i.e. $\langle GET{:}1.GET{:}5.GET{:}8\rangle = \langle GET{:}8.GET{:}1.GET{:}5\rangle$, etc. (see Section 7) for details.

Both the Stack and the Unique Integer can be modeled by deterministic state machines (automata) since in both cases every trace generated describes exactly one state. The difference is that for the Unique Integer traces are built from pairs $(action, response)$ while for the Stack actions alone are sufficient. The case when traces built from actions alone are sufficient will be called *output independent*.

The Drunk Stack is almost the same as the Stack with only one exception, the access program POP is "drunk" so it behaves differently, namely

- POP, if the length of the stack is one then it removes the top of the stack, if the length is greater than 1, POP becomes non-deterministic, it either removes the top of the stack 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 $PUSH$ programs, 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 $PUSH$es are "sober" so they can be used to specify canonical traces.

The use of output independent traces is sufficient for Drunk Stack, however its behaviour cannot be modeled by a deterministic automaton unless the concept of its states is changed (see Section 6).

The Drunk Stack can 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 $PUSH$, the access program TOP is the same as in the Stack. POP behaves as in the Drunk Stack, while the behaviour of $PUSH$ is the following:

- $PUSH(i)$, it either enters an integer i on the stack, or enters two integers i on the stack.

In this case the trace $PUSH(7).PUSH(4)$ leads to $\langle 7,4\rangle$, or $\langle 7,7,4\rangle$, or $\langle 7,4,4\rangle$, 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 the canonical traces interpreted as traces that can unambiguously describe states cannot be defined (the only "sober" action is TOP, but it does not change the states!). We need to proceed differently. One way is to observed 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

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

can 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 [27]. However we reject such an approach. The sets of traces that describe states can be large and complex even for relatively simple (but highly non-deterministic) modules. We believe such approach will result in a complex and unreadable specification. We propose the use of abstract constructor actions instead. In the case of Drunken Stack, all states can easily be specified by an *abstract construction* (invisible) action $push1(i)$ which pushes exactly one i on the stack. The specification obtained is simple and natural (see Section 13, Figures 8 and 9).

We would like to point out that both Drunk Stack and Very Drunk Stack can be represented by deterministic state machines (automata) but then the states can no longer be interpreted as finite sequences of integers. We shall discuss this in detail in Section 6.

3 Alphabet

Since a trace specification describes only those features of a module that are externally observable, the question arises 'What is an atomic observation?'. What constitutes an alphabet from which the traces are built? One may think of two kinds of event occurrences:

- action events
- action-response events

The action events are access program invocations, like $PUSH(5)$ or GET. The set of action events will be denoted by Σ, or Σ_{name} if necessary. For instance

$$\Sigma_{Stack} = \{PUSH(i) \mid\ i \text{ is an } integer\} \cup \{POP, TOP\},\ \text{while } \Sigma_{UniqueInteger} = \{GET\}.$$

The action-response events are access program invocations together with the outputs that are produced. If an access program does not produce any output, it is assumed that it produces the output nil. The set of all *output values* will be denoted by $\mathcal{O}$, or $\mathcal{O}_{name}$ if necessary. It is assumed that $nil \in \mathcal{O}$. The set of all action-response events will be denoted by Δ, with $\Delta \subseteq \Sigma \times \mathcal{O}$. We shall write rather $a{:}d \in \Delta$ instead of $(a,d) \in \Delta$. For example $PUSH(5){:}nil$ or $GET{:}3$ are action-response events, and

$$\Delta_{Stack} = \{PUSH(i){:}nil \mid\ i \text{ is an } integer\} \cup \{TOP{:}i \mid\ i \text{ is an } integer\} \cup \{POP{:}nil\},$$
$$\Delta_{UniqueInteger} = \{GET{:}i \mid\ i \text{ is an } integer\}.$$

Note that the sequences of action-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 action event occurrences only. *If the response is nil it will frequently be omitted*, so if it does not lead to misunderstanding, we shall write just $PUSH(7)$ instead of $PUSH(7){:}nil$.

4 Normal and Exceptional Behaviour

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

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

What kind of a behaviour this trace describes? Clearly this a possible visible behaviour, but $PUSH(i_{n+1})$ represents a *misuse* since the stack may not be pushed beyond its capacity. In other words $t.PUSH(i_{n+1})$ does not represent any *normal behaviour* of the stack module, it represents an *exceptional behaviour* since a misuse is involved. What stack state this trace represent? One solution might be that (because the last push was a misuse) the stack state is not changed, i.e. $t.PUSH(i_{n+1})$ represents the same state as the trace t, namely $\langle i_1, i_2, ..., i_n \rangle$. Another approach might be the last push will cause the bottom (the earliest inserted) element to be "squeezed" out of the stack. In this case $t.PUSH(i_{n+1})$ would represent the state $\langle i_2, i_3, ..., i_{n+1} \rangle$[1]. A trace POP is another example of an exceptional behaviour for empty stack should not be popped, so it is a misuse. A trace TOP is also a misuse and represents an exceptional behaviour, but in this case the problem is what value should be returned, nil, any integer, special value like *undefined*, or something else? In other words, in terms of traces built from action-response events, which trace represents a misuse of TOP at the empty stack: $TOP{:}nil$, $TOP{:}undefined$ or $TOP{:}something\_else$?

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

We shall propose to divide the Trace Assertion Method in two stages, the normal behaviour will be described in the first, main, stage, and an exceptional behaviour may be describe in the second stage. The second stage could be seen as an enhancement of the specification obtained in the first stage.

A behaviour generated by module is *normal* if no misuse is involved, otherwise it is called *exceptional*. The union of normal and exceptional behaviour will be called *visible behaviour*.

The visible behaviour roughly corresponds to *feasible* behaviour of [19, 27]. But the given definition of feasible behaviour is confusing. In principle it says that a trace $a_1{:}d_1.a_2{:}d_2. \ ... \ .a_k{:}d_k$ is feasible iff for each $i = 1, 2, ..., k$, output d_i may be produced by an action a_i after a sequence $a_1{:}d_1.a_2{:}d_2. \ ... \ .a_{i-1}{:}d_{i-1}$. But this means that the trace POP (or $POP{:}nil$) is feasible but TOP (or $TOP{:}something$) may not.

There are cases where the same trace describes both normal and exceptional behaviour. For instance for the Drunk Stack, the trace $PUSH(7).PUSH(4).POP.POP$ describes the normal behaviour under the assumption that the first POP removed only one element from the stack, and it describes an exceptional behaviour when the first POP removed two elements from the

[1] In this case it is not obvious that $t.PUSH(i_{n+1})$ represents an exceptional behaviour. It might be interpreted as a normal behaviour of different kind of stack (compare [36]). What is an exceptional behaviour [29] is a subjective matter, so its role in the mathematical model should be limited.

stack. Then after $PUSH(7).PUSH(4).POP$ the stack is empty and the second POP is a misuse.

The fact that the same trace describes *both* normal and exceptional behaviour *cannot* be specified by any deterministic automaton (see Section 12).

There are traces that do not represent any behaviour. For instance for the Unique Integer module the trace $GET{:}3.GET{:}3$ represents neither normal nor exceptional behaviour. GET is not a misuse at the state described by $GET{:}3$, but it produces non-deterministically any value but 3.

5 Value Functions

In terms of traces the fact that action TOP in the state $\langle 5,7,4\rangle$ produces the value 4 and does not change the state can be expressed in three ways. One way is to say that there is a value function v which with the arguments $PUSH(5).PUSH(7).PUSH(4)$ and TOP returns the value 4, i.e.

$$v(PUSH(5).PUSH(7).PUSH(4), TOP) = 4,$$

and there is a transition function δ_Σ[2] that transforms $PUSH(5).PUSH(7).PUSH(4)$ and TOP into the *one element set*[3] $\{PUSH(5).PUSH(7).PUSH(4)\}$, i.e.

$$\delta_\Sigma(PUSH(5).PUSH(7).PUSH(4), TOP) = \{PUSH(5).PUSH(7).PUSH(4)\}.$$

Another way is to say that there is a transition (extension) function δ that transforms $PUSH(5){:}nil.PUSH(7){:}nil.PUSH(4){:}nil$ and $TOP{:}4$ into $PUSH(5){:}nil.PUSH(7){:}nil.PUSH(4){:}nil$, i.e. (we omit nil's below, $PUSH(i)$ really means $PUSH(i){:}nil$)

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

while $\delta(PUSH(5).PUSH(7).PUSH(4), TOP{:}i) = \emptyset$, if $i \neq 4$.

The third way is to use both the transition function δ and the value function v_δ which returns 4 for the arguments $PUSH(5){:}nil.PUSH(7){:}nil.PUSH(4){:}nil$ and TOP, i.e.

$$v_\delta(PUSH(5){:}nil.PUSH(7){:}nil.PUSH(4){:}nil, TOP) = 4.$$

Note that the function v_δ is redundant since it can be completely derived from δ. The function δ can be seen as a composition of δ_Σ and v. For the stack module the first way seems to be the most appropriate, the second way is acceptable, but the third is rather superfluous which lacks elegance and readability.

For the Unique Integer module the fact that action GET in the state $\{3,6,9\}$ must not produce $i \in \{3,6,9\}$, but it may result in producing the integer 7 and this will change the state

[2] We want the notation used here to be the same as in Section 9, where this problem is formally analyzed.

[3] We prefer $\delta(s,a) = \{t\}$ over $\delta(s,a) = t$ even for deterministic case since we cane use $\delta(s,a) = \emptyset$ to denote that a cannot be normally used at s. We will make use of this in Section 6.

to $\{3,6,7,9\}$. This fact may be expressed in two ways, either using only δ, or δ together with v_δ. The first way is to define

$$\delta(\langle GET{:}3.GET{:}6.GET{:}9\rangle, GET{:}7) = \{\langle GET{:}3.GET{:}6.GET{:}7.GET{:}9\rangle\}$$

and $\delta(\langle GET{:}3.GET{:}6.GET{:}9\rangle, GET{:}i) = \emptyset$, if $i \notin \{3,6,9\}$.

The second way is to define, in addition to δ, a value function v_δ which for these arguments is

$$v_\delta(\langle GET{:}3.GET{:}6.GET{:}9\rangle, GET) = \{i \mid i \notin \{3,6,9\}\}.$$

Surprisingly using δ together with v_δ is very popular in the existing literature [18, 19, 27, 36]. Again, we consider this as superfluous. There may be some cases when an explicit use of v_δ increases readability and precision, but in general it does not. Using δ alone seems to be a better way in most cases.

If the traces are built from action events only then the explicit use of the value function v is a must, δ_Σ alone does not contain any information about the outputs. But if the traces must be built from action-response events (as for Unique Integer module), the function δ alone seems to be sufficient in most cases.

6 Languages and Automata

In this section we show how some fundamental results from automata theory affect the Trace Assertion Method. We *do not* assume that the sets of states consider below are always finite. All the results presented in this section are classical (see for instance [10, 16]).

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 will be denoted by $x.y$.

A language $L \subseteq \Delta^*$ is *prefix closed* if $L = \{x \mid \exists y \in \Delta^*.\ x.y \in L\}$.

6.1 Deterministic and Non-deterministic Automata

By a *deterministic automaton* we mean a quintuple

$$A^{det} = (\Delta, S, \delta, s_0, F),$$

where: Δ is the alphabet, S is the set of states, δ is the transition function, $\delta : S \times \Delta \to S$, s_0 is the initial state, $F \subseteq S$ is the set of final states.

The transition function δ is standardly extended to $\delta^* : S \times \Delta^* \to S$, by defining $\delta^*(s, \varepsilon) = s$, $\delta^*(s, x.a) = \delta(\delta^*(s, x), a)$ for every state s, and every $x \in \Delta^*$, $a \in \Delta$.

A *language recognized* by A^{det} is defined as $L(A^{det}) = \{x \in \Delta^* \mid \delta^*(s_0, x) \in F\}$.

By a *general automaton* we mean a quintuple

$$A = (\Delta, S, \delta, s_0, F),$$

where:Δ is the alphabet, S is the set of states, δ is the transition function, $\delta : S \times \Delta \to 2^S$, s_0 is the initial state, $F \subseteq S$ is the set of final states.

The extended transition function $\delta^* : S \times \Delta^* \to 2^S$ is now defined as $\delta^*(s, \varepsilon) = \{s\}$, and

$$\delta^*(s, x.a) = \bigcup_{y \in \delta^*(s,x)} \delta(y, a)$$

for every state s, and every $x \in \Delta^*$, $a \in \Delta$.

A *language recognized by* A is defined as $L(A) = \{x \in \Delta^* \mid \delta^*(s_0, x) \cap F \neq \emptyset\}$.

The general automata are usually called *non-deterministic* in the literature, however we use the name "general" since "non-deterministic" could cause some confusion later.

Every deterministic automaton can trivially be presented as a general one by defining a new transition function δ' as $\delta'(s, a) = \{s'\}$ if $\delta(s, a) = s'$.

However defining the transition function as $\delta : S \times \Delta \to 2^S$ *has some advantage even if an automaton is de facto deterministic.* If $\delta : S \times \Delta \to S$ then to express the fact that $a \in \Delta$ is never recognized at the state s requires introducing a "dead state" or "sink" since $\delta(s, a) \in S$ for all s and a. When $\delta : S \times \Delta \to 2^S$, the fact that $a \in \Delta$ is never recognized at the state s is just expressed by $\delta(s, a) = \emptyset$, which is very convenient for our purposes.

The general automaton $A = (\Delta, S, \delta, s_0, F)$ is *general and deterministic* if $|\delta(s, a)| \leq 1$ for every $s \in S$, and every $a \in \Delta$. If additionally for every $s \in S$ and every $a \in \Delta$, we have $\delta(s, a) \neq \emptyset$, then A is isomorphic to the deterministic automaton A' which is derived from A by replacing δ by δ', where $\delta'(s, a) = s' \Leftrightarrow \delta(s, a) = \{s'\}$.

Theorem 6.1 *For every general automaton* $A = (\Delta, S, \delta, s_0, F)$ *there is a deterministic automaton* $A^{det} = (\Delta, S^{det}, \delta^{det}, s_0^{det}, F^{det})$ *such that*

1. $L(A) = L(A^{det})$,
2. S *is finite if and only if* S^{det} *is finite.* ■

6.2 Mealy Machines vs Automata

A *Mealy Machine* is a tuple

$$M = (\Sigma, \mathcal{O}, \delta_\Sigma, v, s_0, F)$$

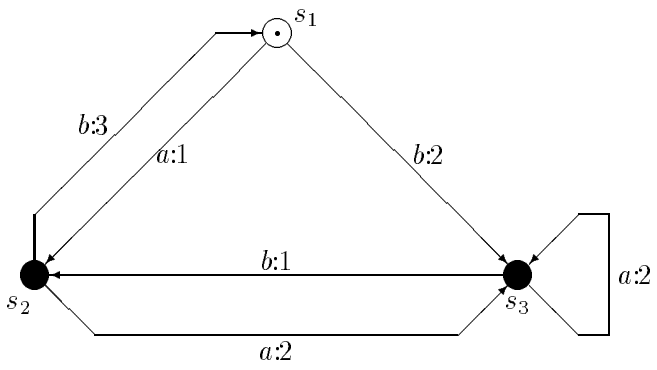
where: Σ is the input alphabet, $\mathcal{O}$ is the output alphabet, δ_Σ is the transition function, $\delta_\Sigma : S \times \Sigma \to 2^S$, v is the output function, $v : S \times \Sigma \to \mathcal{O}$, s_0 is the initial state, and $F \subseteq S$ is the set of final states.

For every Mealy Machine M we can construct the following *general* automaton

$$A_M = (\Delta, S, \delta, s_0, F)$$

where S, s_0, F are the same as in M, $\Delta \subseteq \Sigma \times \mathcal{O}$, $\delta : S \times \Delta \to 2^S$, and (we shall write $a{:}d \in \Delta$ instead of $(a, d) \in \Delta$):

$$\forall a \in \Sigma . \forall d \in \mathcal{O} \;\; a{:}d \in \Delta \;\; \Leftrightarrow \;\; \exists s_1, s_2 \in S.\, s_2 \in \delta_\Sigma(s_1, a) \wedge v(s_1, a) = d,$$



$$\left.\begin{array}{c}\delta_\Sigma(s_1,b)=\{s_3\}\\ v(s_1,b)=2\end{array}\right\} \quad\Leftrightarrow\quad s_1 \xrightarrow{b:2} s_3 \quad\Leftrightarrow\quad \delta(s_1,b{:}2)=\{s_3\}$$

Figure 1: The Mealy machine with $\Sigma = \{a, b\}$, $\mathcal{O} = \{1, 2, 3\}$, or the *general-deterministic* automaton with $\Delta = \{a{:}1, a{:}2, b{:}1, b{:}2, b{:}3\}$. The state s_1 is initial, s_2 and s_2 are final.

$$\forall s \in S. \forall a{:}d \in \Delta.\ \ \delta(s, a{:}d) = \begin{cases} \delta_\Sigma(s,a) & v(s,a) = d \\ \emptyset & v(s,a) \neq d \end{cases}$$

The automaton A_M is equivalent to the Mealy machine M in the sense that every property of M can be described in terms of A_M. Figure 1 illustrates the relationship between M and A_M. We feel that for our purposes using automata instead of Mealy machines makes usually the specification simpler. Furthermore *not every* general (or even general-deterministic) automaton with $\Delta \subseteq \Sigma \times \mathcal{O}$ can be interpreted as a Mealy machine. If to the automaton from Figure 1 one adds a new arrow from s_1 to s_3 labeled by $a{:}2$, the new automaton cannot be interpreted as a Mealy machine.

If all states are final, $F = S$, then we shall omit F and define an automaton as a quadruple $A = (\Delta, S, \delta, s_0)$. In such a case $L(A) = \{x \in \Delta^* \mid \delta^*(s_0, x) \neq \emptyset\}$, and $L(A)$ is prefix closed.

6.3 Right Congruences

An equivalence relation $R \subseteq \Delta^* \times \Delta^*$ is called a right congruence if and only if

$$\forall x, y \in \Delta^*.\ \ xRy \ \ \Leftrightarrow (\forall z \in \Delta^*.\ x.zRy.z).$$

A right congruence R saturates $L \subseteq \Delta^*$, if L is a union of equivalence classes of R.

One can easily prove that if R saturates a prefix closed L then L is a union of all equivalence classes of R except exactly one, i.e., for every $x \in \Delta^* \setminus L$, $[x]_R = \Delta^* \setminus L$.

Let $\approx_L \subseteq \Delta^* \times \Delta^*$ be a relation defined as:

$$\forall x, y \in \Delta^*.\ \ x \approx_L y \ \ \Rightarrow (\forall z \in \Delta^*.\ x.z \in L \Leftrightarrow y.z \in L).$$

The relation $\approx_L$ is the *Nerode equivalence* associated with L on Δ^*. The Nerode equivalence has the following properties.

Proposition 6.2

1. *$\approx_L$ is a right congruence that saturates L.*
2. *If R is a right congruence that saturates L then $R \subseteq \approx_L$, i.e. $xRy \Rightarrow x \approx_L y$.* ■

Given any right congruence R that saturates $L \subseteq \Delta^*$, $A_{R,L}$ is a *deterministic* automaton defined by

$$A_{R,L} = (\Delta, \Delta^*/R, \delta, [\varepsilon]_R, F),$$

where: Δ is the alphabet, Δ^*/R is the set of states, δ is the transition function, $\delta : \Delta^*/R \times \Delta \to \Delta^*/R$, and $\forall x \in \Delta^*. \forall a \in \Delta.\ \delta([x]_R, a) = [x.a]_R$, $[\varepsilon]_R$ is the initial state, F is the set of final states, $F = L/R$, i.e. $F = \{[x]_R \mid x \in L\}$.

The automaton $A_{R,L}$ is a *quotient automaton* defined by R that saturates L. The states of $A_{R,L}$ are the equivalence classes of R, so each state may unambiguously be identified by a single representative of the appropriate equivalence class (*canonical traces!*).

Theorem 6.3 *Let R be a right congruence that saturates L. Then:*

1. *The automaton $A_{R,L}$ accepts the language L, i.e. $L(A_{R,L}) = L$.*
2. *The automaton $A_{R,L}$ is the minimum state automaton accepting L if and only if $R = \approx_L$.*
3. *If L is prefix closed then all states of $A_{R,L}$ except one are final. The only one non-final state is a 'dead state' or 'sink'.*
4. *$xRy \Leftrightarrow \delta^*([\varepsilon]_R, x) = \delta^*([\varepsilon]_R, y)$.* ■

Theorem 6.4

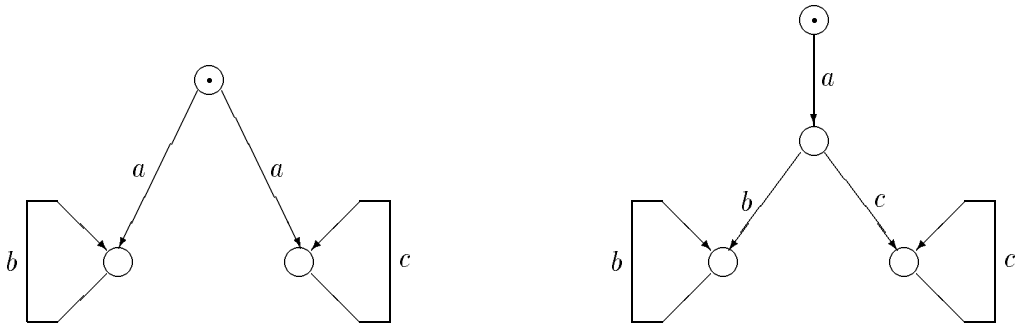
1. *For every deterministic automaton $A = (\Delta, S, \delta, s_0, F)$, the relation $R_A \subseteq \Delta^* \times \Delta^*$, defined as*

$$\forall x, z \in \Delta^*.\ x R_A y \Leftrightarrow \delta(s_0, x) = \delta(s_0, y)$$

   is a right congruence, and the quotient automaton $A_{R_A, L(A)}$ is isomorphic to A.

2. *L is regular if and only if the set $\Delta^*/\approx_L$ is finite.* ■

The above results are valid even if L is not regular and automata are not finite ("minimum" means then "least subset").



General Automaton

General-deterministic Automaton or Deterministic Automaton with the "sink" state not included

Figure 2: Two automata recognizing the language $\varepsilon + ab^* + ac^*$. All states are final.

6.4 Consequences for Trace Assertion Method

There is a single object module interface which is a "black box" that produces a visible behaviour. The behaviour is a set of traces, each trace is a sequence of action-response events. Our task is to provide a readable specification that would describe exactly the behaviour produced by the module.

From Theorems 6.3 and 6.4(1) it follows that in principle the *normal behaviour of every module may be specified by a deterministic automaton* (the relation $\approx_L$ can be constructed for *every* language). The automaton, say A, is minimal if its right congruence R_A equals to $\approx_{L(A)}$. A separate proof of this fact is given in [18] (in a disguised form) while [36] contains separate proofs of some results from Theorem 6.3 (again in a disguised form). No proofs are needed since it all follows from the well established results. Normal behaviour is a prefix closed language (a prefix of a normal trace is also a normal trace), so it could be specified by either deterministic automaton with a "sink" state, or by a non-deterministic automaton with all states being final.

However the results are the existential ones, the provide very little advice how such an automaton may be constructed. Consider a module that has three access programs a, b, and c, each program when invoked displays its name, i.e, a, b, or c, and the normal behaviour of the module is described by the regular expression $\varepsilon + (ab^* + ac^*)$. In other words it is the language $\{\varepsilon, a, a.b, a.b.b, ..., a.c, a.c.c, ...\}$, which is specified by *both* automata in Figure 2.

By observing *only* the normal sequences generated by a module we are *unable to judge if the module is deterministic or not.* Every behaviour may be modeled by a deterministic or general-deterministic automaton, however such a *deterministic model may be extremely complex.*

It is relatively easy to think of such a construction for the Stack and Unique Integer modules. The sequences of integers $\langle i_1, i_2, ..., i_n\rangle$ can serve states for the Stack, and the sets $\{i_1, i_2, ..., i_n\}$ (that can also be described by sequences), can serve as states for the Unique Integer. The transition functions are also *conceptually* easy to describe, for example $\delta(\{1,3,16,34\}, GET: 2) = \{1,2,3,16,34\}$ for the Unique Integer. But the Drunk Stack and Very Drunk Stack are problematic! We know that they can be specified by a deterministic automaton, but to construct such a specification is another problem. Even if constructed such specification is usually very complex and difficult to understand (see [18]). On the other hand the non-deterministic automata based specification is no more difficult to construct and understand than that of the

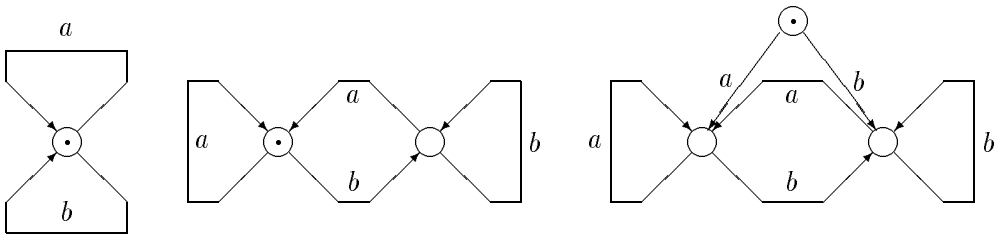


Figure 3: Three deterministic automata recognizing the language $(a+b)^*$. All states are final.

Stack or Unique Integer. How the module is really implemented is beyond our interest, we want to specify the visible behaviour produced by the module, and thinking about the module as a non-deterministic device might make the specification much shorter and easier to understand.

Even for the deterministic modules the choice of automaton is not obvious, and the minimal may not always be the best one. Consider the module with two system calls a and b, and a normal behaviour is specified by the regular expression $(a+b)^*$. All automata from Figure 3 recognize this behaviour. The left automaton is the minimal one, but if a and b are interpreted as $READ$ and $WRITE$ operations, the right automaton might be the best solution. Its states have clear interpretation: *initial, state of reading* and *state of writing*, the automaton seems to be more intuitive one under such interpretation, and three states might help when the specification is going to be refined, for instance exceptional behaviour is added (see Section 12).

We may use both Mealy machines or standard automata as a backbone of the model. The descriptive power of Mealy machines is at best the same as general deterministic automata, only notation is different, more complex in our opinion. It might occasionally be convenient to use Mealy machines instead of standard automata, but in general the standard automata provide 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 [32, 36, 19, 27] Mealy machines were used and we believed that resulted in unnecessary complexity and formal problems [36].

Sets of normal behaviour traces generated by interface modules are prefix closed languages, the general automaton notation has an advantage even in the deterministic case, since we may specify those languages without the "sink" state. We also may omit final states, since prefix closed languages may be modeled by general automata with all states final.

After deciding to use automata as a backbone of the specification technique, we face a very challenging problem: "How to specify the set of states?" When specifying the real interface modules we cannot just say "Let S be a set of states", we have to describe them explicitly, but how? We cannot just list them, they are too big. Some parameterization should help, but we need a methodology first.

7 Defining Objects by Sequences

The ingenuity of the Trace Assertion Method ([3]) was to use traces (i.e. some kind of sequences) not only as a medium to describe behaviour, but to specify states as well. First of all, we observe only traces of actions-responses, so they provide the entire visible information. Sec-

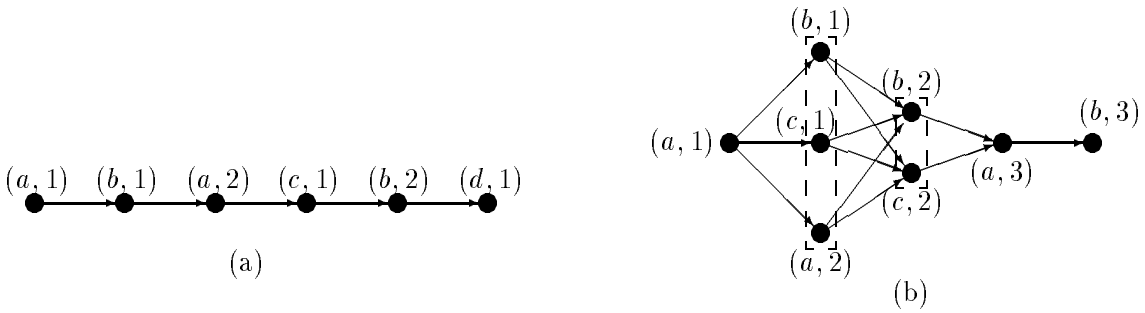


Figure 4: (a) Total order defined by the string $a.b.s.c.b.d$ and (b) Weak order defined by the string $a.\langle a.c.b\rangle.\langle b.c\rangle.a.b$.

ondly, the sequences are easy to specify and understand, and their descriptive power is rather substantial.

Let Δ be an alphabet (possibly infinite), and let $x \in \Delta^*$. What kind of structure x is, and what kind of information x contain? We do not assume any interpretation of elements of Δ.

Consider $x = a.b.a.c.b.d$. The string x can be interpreted as a *total order to*$_x$ of the occurrences of elements of Δ, as illustrated in Figure 4(a). By an occurrence of a we mean a pair (a, i), where i is a natural number indicating the occurrence. What else x can represent?

Consider the string $y = a.b.c.d$. If we have additional information that the order between the occurrences of a, b, c, d does not matter, the string y may represent the set $\{a, b, c, d\}$. Such notation is *de facto* already used in mathematics for almost a century. Note that every set can be interpreted as a partial order with empty ordering relation. The structural difference between $x = a.b.a.c.b.d$ and $y = a.b.c$ is that in y each element of the alphabet occurs at most once, while x contains some repetitions.

The strings where each element of the alphabet occurs at most once will be called *plain strings*. The set of all plain strings over Δ will be denoted by $Plain(\Delta)$.

Let $\langle\cdot\rangle$ denote a *partial operator* such that for every *plain string* $x = a_1.a_2.\ ...\ .a_n$, $\langle x\rangle$ denotes the set (i.e. a special kind of partial order), $\langle x\rangle = \{a_1, ..., a_n\}$. For non-plain strings $\langle\cdot\rangle$ is not defined. Note that for every $a \in \Delta$. $\langle a\rangle = \{a\}$.

By mixing "$\langle\cdot\rangle$" with standard concatenation ".", we can obtain strings like $a.\langle a.c.b\rangle.\langle b.c\rangle.a.b$. Such strings also have been used in mathematics and computer science for years, especially in concurrency theory. They are called *step-sequences* or *subset languages* ([22, 33]), and they represent *weak* (or *stratified*) *partial orders* ([11, 22]). Figure 4(b) illustrates this relationship.

Formally step-sequences are strings 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: $\{a\}.\{a, c, b\}.\{b, c\}.\{a\}.\{b\}$. The latter has a natural interpretation in concurrency theory, where it just describe a sequence of events where some of them are performed simultaneously. However for our purposes, such interpretation is no longer valid, and such a notation may be confused.

From Szpirlajn theorem [11] it follows every partial order corresponds uniquely to the set of all its total extensions. In particular every set $X = \{a_1, ..., a_n\}$ 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 $\{a_1, ..., a_n\}$ can be seen as a description of all *plain* sequences built from a_1,...,a_n. For instance $\{a, b, c\}$ can be seen as a description of the set of strings $\{a.b.c, a.c.b, b.a.c, b.c.a, c.a.b, c.b.a\}$.

In general a step-sequence can 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 $a.\{b.a\}.c.\{a.c\}$ defines the set of sequences: $\{a.b.a.a.c, a.a.b.c.a.c, a.b.a.c.c.a, a.a.b.c.c.a\}$. The set of sequences corresponding to the step-sequence from Figure 4 consists of 12 elements, including for instance $a.a.c.b.b.c.a.b$ and $a.b.c.a.c.b.a.b$

We will keep the name *step sequence*, but will interpret them as a kind of expressions that describes sets of strings.

Formally "our" step-sequences are strings built from the alphabet $\Delta \cup \{\langle, \rangle\}$, such that

- every $x \in \Delta^*$ is a step-sequence,
- if $x \in \Delta^*$ and x is plain, then $\langle x \rangle$ is a step-sequence
- if x and y are step-sequences, then $x.y$ is a step-sequence,
- there are no other step-sequences.

The set of all step-sequences over the alphabet Δ will be denoted by $\langle \Delta^* \rangle$

We define two operations on step-sequences, the first is *concatenation*, denoted standardly by ".", and for instance if $x = \langle a.b \rangle.a.c$ and $y = \langle b.d \rangle.a$, then $x.y = \langle a.b \rangle.a.c.\langle b.d \rangle.a$. The second operation will be called *weak concatenation*, denoted by "$\smile$", and defined as follows. For every $a, b \in \Delta$,

$$a \smile b = \begin{cases} \langle a.b \rangle & \text{if } a \neq b \\ a & \text{if } a = b \end{cases}$$

Similarly, for every plain $t \in \Delta^*$ (ε is plain),

$$\langle t \rangle \smile a = \begin{cases} \langle t.a \rangle & \text{if } t.a \text{ is plain} \\ \langle t \rangle & \text{if } t.a \text{ is not plain} \end{cases}$$

Every $x, y \in \langle \Delta^* \rangle \setminus \{\varepsilon\}$ can be represented as $x = x_1.\alpha$, $y = \beta.y_1$, where $x_1, y_1 \in \langle \Delta^* \rangle$, $\alpha = \langle t \rangle$ or $\alpha = a$, $\beta = \langle s \rangle$ or $\beta = b$, t and s are plain, $a, b \in \Delta^*$, we may define $x \smile y$ in the following way:

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

Finally we assume that $x \smile \varepsilon = x$ and $\varepsilon \smile y = y$.

In principle $\smile$ consists in merging the last "step" of x with the first "step" of y.

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 $(a.\langle a.b \rangle) \smile (\langle c.d \rangle.a) = a.\langle a.b.c.d \rangle.a$. If both x and y are plain sequences then $\langle x \rangle \smile \langle y \rangle = \langle x.y \rangle$ which corresponds exactly to building set theory union of elements from x and y. For example $\langle a.b \rangle \smile \langle b.c.d.e \rangle = \langle a.b.c.d.e \rangle$ corresponds to $\{a, b\} \cup \{b, c, d, e\} = \{a, b, c, d.e\}$.

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 $y = x \smile z$.

The semantics of "our" step-sequences can be defined as 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. $\forall x \in Plain(\Delta).\ sem(\langle x \rangle) = \{y \mid set(x) = set(y)\}$,
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 strings (see [16]). For instance $sem(\langle a.b.c \rangle) = \{a.b.c, a.c.b, b.a.c, b.c.a, c.a.b, c.b.a\}$, $sem(a.\langle b.a \rangle.c.\langle a.c \rangle) = \{a.b.a.c.a.c, a.a.b.c.a.c, a.b.a.c.c.a, a.a.b.c.c.a\}$.

Two step sequences x, y are considered *equal*, $x = y$, if and only if $sem(x) = sem(y)$. The symbol $=$ has here the same meaning as when for instance we write $(a + b)c = ac + bc$ for arithmetic or regular expressions.

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 will use step-sequences to specify states of automata.

8 Trace Only Automata

From here on by an automaton we shall mean a general automaton with all states final, so the set F will not appear in any definition.

Let $A = (\Delta, S, \delta, s_0)$ be an automaton. We shall say that A has *canonical trace property* (*ct-property*) if for every state $s \in S$ there is a trace $x_s \in \Delta^*$ such that $\delta^*(s_0, x_s) = \{s\}$. Not every automaton has ct-property and every (general) deterministic automaton has ct-property. If we take automaton from the left hand side of Figure 2 and erase arrows labeled by c and b, the resulting automaton does not have ct-property. Frequently there is more than one x_s satisfying $\delta^*(s_0, x_s) = \{s\}$. For example for the automaton from Figure 1 and the state s_3 we have (s_1 is initial here) $\delta^*(s_1, b.2.(a.2)^i) = \{s_3\}$, for every $i = 0, 1, ..., \infty$.

If A has ct-property we may define a set of *canonical traces* (see [32]). A set of traces $CanTr \in \Delta^*$ is *canonical* if for every $s \in S$ there is exactly one x_s in $CanTr$ such that $\delta^*(s_0, x_s) = \{s\}$.

If $A = (\Delta, S, \delta, s_0)$ has ct-property, and $CanTr$ is any set of canonical traces of A, and for every state $s \in S$, its unique representation in $CaTr$ is denoted by x_s, then A is isomorphic to $A^{ct} = (\Delta, CanTr, \delta^{ct}, x_{s_0})$, where $\delta^{ct}(x_s, a) = \delta(s, a)$.

The automata like A^{ct} may be called *trace only automata* since their states are defined in terms of traces as well. Mealy machine counterparts of trace only automata have been extensively used in the trace assertion method related literature ([17, 18, 19, 27, 32, 36], etc.). The problem is that using traces may frequently result in a kind of asymmetry which makes

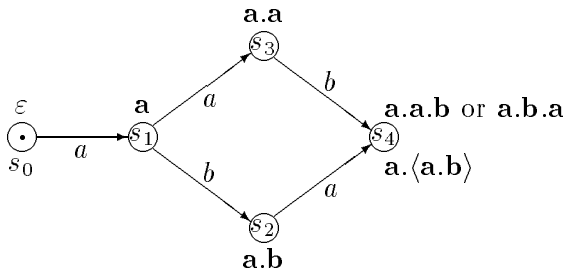


Figure 5: Example of asymmetry when canonical traces are used.

the specification less readable than expected. Consider an automaton in Figure 5. It occurs very often as a 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 $\delta^{ct}(x, b) = \{x.b\}$. The canonical trace $y = a.b$ is not a prefix of $a.a.b$, so $\delta^{ct}(y, a) \neq \{y.a\}$. The asymmetry is induced by the choice of a canonical trace, the automaton itself is symmetrical, from the state s_1 we will reach s_4 in two steps, using both a and b in any order, $a.b$ or $b.a$. Such asymmetry make some specifications unnecessary complex. The Unique Integer module is a classical example, but the problem occurs in many real interface modules 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 $a.\langle a.b\rangle$ ($a.\langle a.b\rangle = (a.b) \smile a = (a.a) \smile b$).

When an automaton A has a ct-property, we may define the set of *canonical step-traces*, $\mathcal{C} \subseteq \langle \Delta^* \rangle$, as any subset of $\langle \Delta^* \rangle$ satisfying:

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

The symbol $\exists!$ denotes "there exists exactly one". Note that 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 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 we must solve. But if the best and most readable model for a behaviour generated by a given interface module is an automaton-like structure without ct-property, we can use a concept similar to *labeled transition system* [2]. The difference between automata and transition systems is that in transition systems each arrow has unique name. The elements of Δ attached to arrows in automata are called *labels* in transition systems. Hence when we use the name "labeled transition system" we mean that each arrow has *two* attachments, a unique name, and not necessary unique label. We do not need each arrow to be unique, we need only ct-property, so the following construction is proposed.

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

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

where Δ is the alphabet, Υ is the *state constructors* alphabet, S is the set of states, δ is the transition function, and $\delta : (S \times \Delta) \cup (S \times \Upsilon) \rightarrow 2^S$, s_0 is the initial state. The transition function δ must satisfy the following conditions:

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

The language generated by the automaton with the alphabet of states constructors A is given by:

$$L(A) = \{x \in \Delta^* \mid \delta^*(t_0, x) \neq \emptyset\}$$

.

The elements of Υ does not occur in $L(A)$. We *do not* assume $\Delta \cap \Upsilon = \emptyset$. The first condition says that A restricted to Υ is a (general) deterministic automaton. Hence the ct-property is guaranteed. The second condition guarantees that each arrows 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-traces.

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

We shall base the trace assertion techniques on the trace only automata described in this section. Trace Assertion Specifications defined formally below are just a more specific and extended versions of the traces-only automata defined in this section.

9 Trace Assertion Specifications

The traces-only automata as those considered in previous sections are a little bit too restricted to model all aspects of trace assertion technique. We need to be able to express the difference between action alphabet and action-response alphabet, between normal and exceptional behaviour, to exploit a problem of output values etc. Those extended automata-like structures will be called trace assertion specifications.

By a *trace assertion specification* we mean a tuple

$$TA = (\Sigma, \mathcal{O}, \Delta, \mathcal{C}, \delta, t_0),$$

where:

Σ is the set of *actions*,
$\mathcal{O}$ is the set of *output values*, $nil \in \mathcal{O}$,
$\Delta \subseteq \Sigma \times \mathcal{O}$ is the set of *action-responses*,
$\mathcal{C} \subseteq \langle \Delta^* \rangle$ is the set of *canonical step-traces*,
$\delta : \mathcal{C} \times (\Sigma \times \mathcal{O}) \rightarrow 2^{\mathcal{C}}$ is the *transition* (or *extension*) *function*,
$t_0 \in \mathcal{C}$ is the *initial canonical step-trace*,
and the following are satisfied:

1. $\forall t \in \mathcal{C}. \forall x \in sem(t). \, \delta^*(t_0, x) = \{t\}$,
2. $\forall t \in \mathcal{C}. \forall a \in \Sigma. \forall d \in \mathcal{O}. \, a{:}d \notin \Delta \Leftrightarrow \delta(t, a{:}d) = \emptyset$.

The first condition says that every canonical step-trace t describes unambiguously one state, and this is the state the sequence $x \in sem(t)$ leads to. The second condition enables the mapping δ to be formally defined outside the set Δ (δ has no interpretation outside Δ).

We shall write rather $\delta(c, a{:}d)$ than $\delta(c,(a,d))$. Note that without loss of generality we may (and frequently will) identify $\delta : \mathcal{C} \times (\Sigma \times \mathcal{O}) \to 2^{\mathcal{C}}$ with $\delta : \mathcal{C} \times \Delta \to 2^{\mathcal{C}}$.

In all examples, if $d = nil$, we would rather write a instead of $a{:}nil$. For instance, if it would not lead to a misunderstanding, we would rather write $PUSH(i)$ instead of $PUSH(i){:}nil$.

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, for both the Stack and the Less Drunken Stack:
$\delta(PUSH(5).PUSH(7).PUSH(4), TOP{:}4) = \{PUSH(5).PUSH(7).PUSH(4)\}$,
$\delta(PUSH(5).PUSH(7).PUSH(4), TOP{:}8) = \emptyset$,
$\delta(PUSH(5).PUSH(7).PUSH(4), PUSH(5)) = \{PUSH(5).PUSH(7).PUSH(4).PUSH(5)\}$.
The access program called POP behaves differently in the Stack than in the Drunk Stack, and for example:
$\delta(PUSH(5).PUSH(7).PUSH(4), POP) = \{PUSH(5).PUSH(7)\}$
for the Stack, while
$\delta(PUSH(5).PUSH(7).PUSH(4), POP) = \{PUSH(5).PUSH(7), PUSH(5)\}$
for the Drunk Stack.

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:
$\delta(GET{:}3.GET{:}6.GET{:}9, GET{:}7) = \{\langle GET{:}3.GET{:}6.GET{:}9.GET{:}7\rangle\} =$
$\{\langle GET{:}3.GET{:}6.GET{:}7.GET{:}9\rangle\}$,
$\delta(GET{:}3.GET{:}6.GET{:}9, GET{:}3) = \emptyset$.

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:

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

The notion of competence function follows from [27]. 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 the Stack and the Less Drunken 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 had been used.

The trace assertion specification TA describes the following *normal behaviour*:

$$NB(TA) = \{x \in \Delta^* \mid \delta^*(t_0, x) \neq \emptyset\}.$$

The normal behaviour is just a set of all strings recognized by TA treated as an automaton with all states being final.

The *reduction* function considered in [32, 36] and others can here be defined as $red : \Delta^* \to 2^{\mathcal{C}}$, and $red(x) = \delta^*(t_0, x)$ for all $x \in \Delta^*$.

Let $\pi : \langle \Delta^* \rangle \to \langle \Sigma^* \rangle$ be a projection mapping defined by:

$$\begin{array}{l} \pi(\varepsilon) = \varepsilon, \\ \forall a{:}d \in \Delta.\ \pi(a{:}d) = a, \\ \forall x, y \in \langle \Delta^* \rangle.\ \pi(x.y) = \pi(x).\pi(y), \\ \forall x \in \langle \Delta^* \rangle.\ \pi(\langle x \rangle) = \langle \pi(x) \rangle \end{array}$$

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 and only if for every $x, y \in NB(TA)$,

$$x = y \quad \Leftrightarrow \quad \pi(x) = \pi(y).$$

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

A trace assertion specification is *output dependent* if it is not output independent.

Both the Stack and the Drunk Stack are output independent while Unique Integer is not. *In [32, 36] and many others the output independent trace assertion specifications are called deterministic while output dependent are called non-deterministic. We shall not use these names since the words determinism and non-determinism have different meaning in automata theory* (compare [10, 16]).

A trace assertion specification is *deterministic* if

$$\forall t \in \mathcal{C}.\ \forall a \in \Delta.\ |\delta(t, a)| \leq 1.$$

A trace assertion specification is *non-deterministic* if it is not deterministic.

Both the Stack and Unique Integer are deterministic, while the Less Drunken Stack is not. Determinism does not imply output independence and output independence does not imply determinism. The Unique Integer is deterministic but output dependent, the Drunk Stack is non-deterministic but output independent.

10 Mealy Forms of Trace Assertion specification

If TA is output independent (as for the Stack module), it may be represented as a kind of a Mealy machine, with separate specification of output function. The most trace assertion models in literature are based on Mealy machines. We think that, in general, the general automata concept is better, but for the output independent TA's Mealy model also leads to equally readable specification. It also helps to explain the relationship with the algebraic specification (see Chapter 15).

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

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

Proof. Suppose that there are $d, d' \in \Delta^*$, $a \in \mathcal{C}$, such that $d \neq d'$, and $\delta(t, a{:}d) \neq \emptyset$, $\delta(t, a{:}d') \neq \emptyset$. 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 $\delta(t, a{:}d)$ is not empty.

Given an *output independent* trace assertion specification, we define the mapping $\delta_\Sigma : \pi(\mathcal{C}) \times \Sigma \to 2^{\pi(\mathcal{C})}$, the *actions only transition function*, as

$$\forall t \in \mathcal{C}.\, \forall a \in \Sigma.\, \delta_\Sigma(\pi(t), a) = \pi(\delta(t, a{:}d))$$

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

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

Lemma 10.1 guarantees the correctness of the above definitions.

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

$$\delta(t, a{:}d) \in s \;\; \Leftrightarrow \;\; \delta_\Sigma(\pi(t), a) \in \pi(s) \wedge v(\pi(t), a) = d.$$

Proof. ($\Rightarrow$) From the definitions of δ_Σ and v.
($\Leftarrow$) Suppose $s \notin \delta(t, a{:}d)$. We have to consider two cases.
Case 1. $\delta(t, a{:}d) = \emptyset$. From the definition of v, we have $v(\pi(t), a) = nil \neq d$.
Case 2. $\delta(t, a{:}d) \neq \emptyset$. Then $\delta_\Sigma(\pi(t), a) = \pi(\delta(t, a{:}d))$. If $\pi(s) \in \delta_\Sigma(\pi(t), a)$ 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 δ is completely defined by δ_Σ and v.

The Stack is output independent, so instead of
$\delta(PUSH(5).PUSH(7).PUSH(4), TOP{:}4) = \{PUSH(5).PUSH(7).PUSH(4)\}$,
one can write equivalently:
$\delta_\Sigma(PUSH(5).PUSH(7).PUSH(4), TOP) = \{PUSH(5).PUSH(7).PUSH(4)\}$, and
$v(PUSH(5).PUSH(7).PUSH(4), TOP) = 4$.
The latter convention is widely used in the existing literature.

Proposition 10.2 allows us to represent any output independent trace assertion specification in an equivalent form, which will be called Mealy form.

By a *Mealy form* of a trace assertion specification TA we mean a Mealy machine with all states final

$$TA^{Mealy} = (\Sigma, \mathcal{O}, \pi(\mathcal{C}), \delta_\Sigma, v, t_0).$$

For every trace assertion specification, not necessary output independent, TA, an explicate value function $v_\delta : \mathcal{C} \times \mathcal{O} \to 2^{\mathcal{O}}$ can be defined as $v_\delta(t,a) = \{d \mid \delta(t, a{:}d) \neq \emptyset\}$.

But v_δ differs from v. The function v can only be defined for an output independent TA, occurs together with δ_Σ and cannot be derived form δ_Σ. The mapping v_δ is redundant, it is derived from δ.

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

$\delta(\langle GET{:}3.GET{:}6.GET{:}9\rangle, GET{:}7) = \{\langle GET{:}3.GET{:}6.GET{:}9.GET{:}7\rangle\}$, and

$\delta(\langle GET{:}3.GET{:}6.GET{:}9\rangle, GET{:}i) = \emptyset$ if $i \in \{3,6,9\}$,

or one may write equivalently:

$\delta(\langle GET{:}3.GET{:}6.GET{:}9\rangle, GET{:}7) = \{\langle GET{:}3.GET{:}6.GET{:}7.GET{:}9\rangle\}$, and

$\delta(\langle GET{:}3.GET{:}6.GET{:}9\rangle, GET{:}i) = \emptyset$ if $i \in \{3,6,9\}$,

$v_\delta(\langle GET{:}3.GET{:}6.GET{:}9\rangle, GET) = \{i \mid i \notin \{3,6,9\}\}$.

The second way is longer and, in our opinion, does not increase readability. It is used in [36] and others.

All the concepts introduced so far do not allow to model the 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 action-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

By a *trace assertion specification with State Constructors* we mean a tuple

$$\Upsilon TA = (\Sigma, \mathcal{O}, \Delta, \Upsilon, \mathcal{C}, \delta, t_0),$$

where: Σ is the set of *actions*, $\mathcal{O}$ is the set of *output values*, $nil \in \mathcal{O}$, $\Delta \subseteq \Sigma \times \mathcal{O}$ is the set of *action-responses*, $t_0 \in \mathcal{C}$ is the *initial canonical step-trace*, exactly as for Trace Assertion Specification, and

Υ is the set of *state constructors*,

$\mathcal{C} \subseteq \langle \Upsilon^* \rangle$ is the set of *canonical step-traces*,

$\delta : \mathcal{C} \times ((\Sigma \times \mathcal{O}) \cup \Upsilon) \to 2^{\mathcal{C}}$ is the *transition* (or *extension*) *function*,

and the following rules (same as in the definition of TA) are satisfied:

1. $\forall t \in \mathcal{C}.\, \forall x \in sem(t).\, \delta^*(t_0, x) = \{t\}$,
2. $\forall t \in \mathcal{C}.\, \forall a \in \Sigma.\, \forall d \in \mathcal{O}.\, a{:}d \notin \Delta \Leftrightarrow \delta(t, a{:}d) = \emptyset$.

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

For the Very Drunk Stack module, the set Υ is the set of all abstract invisible actions $push1(i)$, 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 $\Delta \cap \Upsilon = \emptyset$.

For instance

$\delta(push1(5).push1(7).push1(4), TOP{:}4) = \{push1(5).push1(7).push(4)\}$

$\delta(push1(5).push1(7).push1(4), TOP{:}8) = \emptyset$.

$\delta(push1(5).push1(7).push1(4), PUSH(5)) = \{t_1, t_2\}$,

where $t_1 = push1(5).push1(7).push1(4).push1(5)$,
$t_2 = push1(5).push1(7).push1(4).push1(5).push1(5)$, and
$\delta(push1(5).push1(7).push1(4), POP) = \{push1(5).push1(7), push1(5)\}$.

The trace assertion specification with state constructors ΥTA defines the following normal behaviour

$$NB(\Upsilon TA) = \{x \in \Delta^* \mid \delta^*(t_0, x) \neq \emptyset\}.$$

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

Introducing invisible state constructors is clearly *against* the philosophy of 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 behaviour of a module in the most easy and readable yet precise way. The states are auxiliary concepts, and the invisible actions seem to serve well as the state constructors.

12 Enhancements and Full Trace Assertion Specifications

The model presented above defines only the normal behaviour. It does not touch the exceptional behaviour at all. If $\kappa(t, a) = 0$ then for all $d \in \mathcal{O}$, we have $\delta(t, a{:}d) = \emptyset$, which means that a at t is a misuse and it does not generate any normal behaviour. We cannot forbid it since there is a principal assumption [29], which basically say that: " For every module every sequence of access programs is allowed" (but not every sequence of action-responses!).

Formally it means that if VB is the full visible behaviour of a given module and Σ is its action alphabet, then $\pi(VB) = \Sigma^*$. Hence if $\kappa(t, a) = 0$ then some exceptional behaviour may be defined. Since the exceptional behaviour had not been used to define the principal structure of the trace assertion specification, it is added as a kind of enhancement or refinement. Let $TA = (\Sigma, \mathcal{O}, \Delta, \mathcal{C}, \delta, t_0)$ be a trace assertion specification. In principle, an enhancement of TA consists in defining new δ_E such that $\delta_E(t, a{:}d) \neq \emptyset$ when $\kappa(t, a) = 0$. It is a structure complimentary to TA.

By an enhancement $enh(TA)$ of TA we mean a quintuple

$$enh(TA) = (\Sigma_E, \mathcal{O}_E, \Delta_E, \mathcal{C}_E, \delta_E),$$

where: Σ_E as an *enhanced set of actions*, $\mathcal{O}_E$ is an *enhanced set of output values*, Δ_E is an *enhanced action-response alphabet*, $\mathcal{C}_E \subseteq \langle \Delta_E^* \rangle$ is an *enhanced set of canonical traces*, $\delta_E : \mathcal{C} \times \Delta_E \to 2^{\mathcal{C}_E}$ is an *enhanced transition function*, which satisfies

1. $\forall t \in \mathcal{C}.\, \forall a \in \Sigma.\, \kappa(t, a) = 0 \quad \Leftrightarrow \quad (\exists! d \in \mathcal{O}_E.\, \delta_E(t, a{:}d) \neq \emptyset)$.
2. $\forall t \in \mathcal{C}_E.\, \forall a \in \Sigma_E.\, \forall d \in \mathcal{O}_E.\, |\delta_E(t, a{:}d)| \leq 1$.

The condition 1 says that δ_E is defined for misuses only, and that it is always output independent. The second condition states that the enhancement is always deterministic.

The enhancement $enh(TA)$ is called *plain* if $\Sigma_E \subseteq \Sigma$, $\Delta_E \subseteq \Delta$, and $\mathcal{C}_E \subseteq \mathcal{C}$. 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 the Stack and the Drunk Stack a plain enhancement can be defined by:

$\delta_E(\varepsilon, POP) = \delta_E(\varepsilon, TOP{:}nil) = \{\varepsilon\}$,

$\delta_E(t_{full}, PUSH(i)) = \{t_{full}\}$,

where t_{full} is the canonical step-trace corresponding to the full stack, and $\delta_E(t, a{:}d) = \emptyset$ for the rest of t, a, and d. For the Unique integer the enhancement can be defined by:

$\delta_E(t_{all}, GET{:}nil) = \{t_{all}\}$

where t_{all} is the canonical trace corresponding to the state where all available integers are used up, and $\delta_E(t, a{:}d) = \emptyset$ for the remaining t, a, and d.

Since every enhancement is output independent it can be represented in a Mealy form $(\Sigma_E, \mathcal{O}_E, \pi(\mathcal{C}), \delta_{\Sigma_E}, v_E)$. The definition is practically identical as for output independent TA's. The only difference is that the enhancements do not posses initial step-traces.

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

$\delta_{\Sigma_E}(\varepsilon, POP) = \delta_{\Sigma_E}(\varepsilon, TOP) = \{\varepsilon\}$,

$\delta_{\Sigma_E}(t_{full}, PUSH(d)) = \{t_{full}\}$,

$v_E(t_{full}, TOP) = nil$,

and $\delta_{\Sigma_E}(t, a) = \emptyset$ for the rest of t and a.

The full specification is just a union of TA and $enh(TA)$.

By a *full trace assertion specification* we mean a composition (union) of TA and $enh(TA)$, $FTA = TA \cup enh(TA)$, i.e.

$$FTA = (\Sigma \cup \Sigma_E, \mathcal{O} \cup \mathcal{O}_E, \Delta \cup \Delta_E, \mathcal{C} \cup \mathcal{C}_E, \delta \cup \delta_E, t_0).$$

One may easily show that $\delta_F = \delta \cup \delta_E$ satisfies $\delta_F : (\mathcal{C} \cup \mathcal{C}_E) \times (\Delta \cup \Delta_E) \to 2^{\mathcal{C} \cup \mathcal{C}_E}$, and $\forall t \in \mathcal{C} \cup \mathcal{C}.\, \forall a \in \Delta \cup \Delta_E.\, \forall d \in \mathcal{O} \cup \mathcal{O}_E$.

$$\delta_F(t, a{:}d) = \begin{cases} \delta(t, a{:}d) & \text{if } t \in \mathcal{C} \wedge \kappa(t, a) = 1 \\ \delta(t, a{:}d) & \text{if } (t \in \mathcal{C} \wedge \kappa(t, a) = 0) \vee t \notin \mathcal{C} \end{cases}$$

The full trace assertion specification FTA describes the following *visible behaviour*

$$VB(FTA) = \{x \in (\Delta \cup \Delta_E)^* \mid \delta_F^*(t_0, x) \neq \emptyset\}.$$

Note that $\pi(VB(FTA)) = (\Sigma \cup \Sigma_E)^*$ for *every* FTA.

The *exceptional behaviour* described by FTA, $EB(FTA)$, is defined as follows:

$$x \in EB(FTA) \;\Leftrightarrow\; x \in VB(FTA) \wedge (\exists y, z, a, d.\, x = y.a{:}d.z \wedge \forall t \in \delta^*(t_0, y).\, \kappa(t, a) = 0).$$

In other words $x \in VB(FTA)$ describes an exceptional behaviour if there is a at least one path from in FTA that starts from t_0, is labeled by x and involves a misuse. For instance $POP.POP.PUSH(2)$ describes an exceptional behaviour for the Stack module. Here we have $\delta(\varepsilon, \varepsilon) = \{\varepsilon\}$ and $\kappa(\varepsilon, POP) = 0$.

Proposition 12.1

1. $VB(FTA) = NB(TA) \cup EB(FTA)$.

2. *If TA is deterministic then*
   $NB(TA) \cap EB(FTA) = \emptyset$ *and* $EB(FTA) = VB(FTA) \setminus NB(TA)$.

Proof. (1) Clearly $NB(TA) \subseteq VB(FTA)$. Let $x \in EB(FTA)$. This means $x = y.a:d.z$ and there is $t \in \delta(t_0, y)$ such that $\delta_F^*(t, a:d.z) \neq \emptyset$. Hence $\delta_F^*(t, a:d.z) \subseteq \delta_F^*(t_0, x)$, i.e. $\delta_F^*(t_0, x) \neq \emptyset$ which means that $x \in VB(FTA)$. Thus $NB(TA) \cup EB(FTA) \subseteq VB(FTA)$. Let $x = a_1{:}d_1. \ ... \ .a_k{:}d_k \in VB(FTA)$. This means there is a sequence $t_0, t_1, ..., t_k$ of elements of $\mathcal{C} \cup \mathcal{C}_F$ such that $t_i \in \delta_F(t_{i-1}, a_i)$ for $i = 1, 2, ..., k$. There may exist more than one such sequence. If at least one such sequence has the property: $\exists\, 0 \leq j \leq k \Leftrightarrow 1.\ \kappa(t_{j-1}, a_j) = 0$, then $x \in EB(FTA)$, if no sequence has such property then $x \in NB(TA)$.
(2) Let $x \in NB(TA) \cap EB(FTA)$. Then $x = y.a{:}d.z$ and there are $t, t' \in \delta^*(t_0, y)$ such that $\kappa(t, a) = 0$, $\kappa(t', a) \neq 0$, so $|\delta^*(t_0, y)| \geq 2$, i.e. TA is not deterministic. ∎

Note that for instance for the Drunk Stack: $PUSH(3).PUSH(6).POP.POP \in NB(TA) \cap EB(FTA)$.

The *output independence* and *determinism* is for FTA defined exactly in the same way as for TA. the only difference is using δ_F and VB instead of δ and NB. From Proposition 12.1 it follows that if $NB(TA) \cap EB(FTA) \neq \emptyset$ then the full behaviour cannot be modeled by a deterministic trace assertion specification.

For every output independent FTA we can standardly built its Mealy form FTA^{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 will be 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 module access-program names, the number of arguments each program takes, the type of each argument, and the type of each return value. In the standard form it also specifies action-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. [32, 36]), however in majority of (well thought of) cases this is a relatively simple formula. It turns out the sequences have a lot of description power indeed. The convention

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

as a shorthand for $E(x_j).E(x_{j+1}). \ldots .E(x_k)$ 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

$$\delta(t, a{:}d) = \{t_1, ..., t_k\}$$

for all actions a defined in the Syntax section. The traces $t, t_1, ..., t_k$ are the canonical step-traces. If $d = nil$, then we will write a instead of $a:d$ (the same applies to the Canonical Step-trace Definition section). Since δ is a total function it must be defined for every possible t, a and d. The convention that empty sets are specified by omission will be used. If for a particular triple c, a and d, the value of the function $\delta(t, a:d)$ *does not appear in the Trace Assertions section this means that* $\delta(t, a{:}d) = \emptyset$.

In the *Mealy form* the Trace Assertions section is a sequence of trace assertions of the form $\delta_\Sigma(t, a) = \{t_1, ..., t_k\}$, with the symbol Σ usually omitted.

To specify the deterministic trace assertions the following tabular notation is used[4].

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

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

The column *Equivalence* defines the canonical step-trace t' such that $\delta(t, a{:}d) = \{t'\}$. 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* $\delta(t, a{:}d) = \{$this_c'$\}$. The columns *Conditions* and *Trace Patterns* can be omitted if not needed. The empty cells in those columns will denote the predicate *true*.

For the *non-deterministic* trace assertion the tabular notation is slightly different, namely:

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

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)	$t_{2,1}$		$t_{2,2}$	
.....	.....	.....			

[4] See [21, 23, 30] for details on tabular notation.

In this case the rows should be read as follows:
*if condition*1 *and pattern*1(t) *then* $\delta(t, a{:}d) = \{t_{1,1}, t_{1,2}, t_{1,3}, t_{1,4}\}$,
*if condition*2 *and pattern*2(t) *then* $\delta(t, a{:}d) = \{t_{2,1}, t_{2,2}\}$,
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.

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 *Full* 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 shall use the convention that all the rows added by the enhancement are marked by the symbol "%" at the beginning.*

Figures 6, 7, 8, 9, 10 present various forms of Trace Assertion Specifications of the Stack, Drunk Stack and Very Drunk Stack modules in the specification format described above.

14 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 more (any number in general) 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 $stack\_name$ (c.f. [6]) or may require object generator like $new(stack\_name)$. A very 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[5]. We need only an apparatus to make the states of individual objects distinct, to transform global states by both global actions (like *concatenate*, which affects more than one individual state, or new, which create a new local state), and local actions (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 $\{stack1\sharp PUSH(3).PUSH(5), stack2\sharp PUSH(3).PUSH(1).PUSH(8), stack3\sharp\varepsilon\}$ may represent a global state consisting of three stacks $stac1$, $stack2$, $stack3$, 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*

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

Syntax of Access Programs

Name	Argument	Value	Action-response Forms
POP			POP:nil
$PUSH$	$integer$		$PUSH(d)$:nil
TOP		$integer$	TOP:d

Canonical Step-traces

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

$t_0 = \varepsilon$

Trace Assertions

$\delta(t, POP) =$

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

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

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

$\delta(t, TOP{:}d) =$

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 6: Full Trace Assertion Specification for Stack Module

Syntax of Access Programs

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

Canonical Step-traces

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

$t_0 = \varepsilon$

Trace Assertions

$\delta(t, POP) =$

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

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

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

$\delta(t, TOP) =$

Equivalence
t

or

$\delta(t, TOP) =$

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

Values

$v(t, TOP) =$

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 7: Mealy Form of the Full Trace Assertion Specification for Stack Module

Syntax of Access Programs

Name	Value	Action-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

$\delta(t, GET{:}d) =$

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

Dictionary

$limit$: the number of available integers $limit = maxinteger \Leftrightarrow mininteger + 1$
$maxinteger$: the maximum available integer
$mininteger$: the minimum available integer
$length(t)$: the length of the trace t

Figure 8: Full Trace Assertion Specification for Unique Integer Module

sets which is discussed in the next chapter.

14.1 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 $\alpha \sharp x \in \mathcal{L} \times X$ instead of $(\alpha, x) \in \mathcal{L} \times X$. If $\mathcal{L} = \{1, 2, 3\}$, $X = \{a, b\}$ then $\{1\sharp a, 1\sharp b, 2\sharp a\}$ is an example of a labeled set.

A set $\mathcal{X} \subseteq \mathcal{L} \times X$ is *uniquely labeled* if and only if

$$\forall \alpha \in \mathcal{L}. \forall x, y \in X. (\alpha \sharp x \in \mathcal{X} \ \wedge \ \alpha \sharp y \in \mathcal{X}) \quad \Leftrightarrow \quad x = y.$$

$\mathcal{X}$ is uniquely labeled if every element of it has an unambiguous label. For example $\{1\sharp a, 2\sharp a\}$ is uniquely labeled, while $\{1\sharp a, 1\sharp b, 2\sharp a\}$ 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

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

The set $\mathcal{L}(\mathcal{X})$ is the set of all labels used to labeled elements of $\mathcal{X}$. For instance $\mathcal{L}(\{1\sharp a, 2\sharp b\}) = \{1, 2\}$.

Syntax of Access Programs

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

Canonical Step-traces

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

$t_0 = \varepsilon$

Trace Assertions

$\delta(t, POP) =$

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

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

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

$\delta(t, TOP$:$d) =$

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 9: Full Trace Assertion Specification for Drunk Stack Module

Syntax of Access Programs

Visible Name	Abstract Name	Argument	Value	Action-response Forms
POP				POP:nil
$PUSH$		$integer$		$PUSH(d)$:nil
TOP			$integer$	TOP:d
	$push1$	$integer$		$push1(d)$

Canonical Step-traces

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

$t_0 = \varepsilon$

Trace Assertions

$\delta(t, POP) =$

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

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

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

$\delta(t, TOP{:}d) =$

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 10: Full Trace Assertion Specification for Very Drunk Stack Module

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

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

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

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

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

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 $\mathcal{X} \hookleftarrow \mathcal{Y} = \mathcal{Y}$ and $\mathcal{X} \oplus \mathcal{Y} = \mathcal{Y} \setminus \mathcal{X}$. 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 $\{1\sharp a, 2\sharp b\} \hookleftarrow \{1\sharp b\} = \{1\sharp b, 2\sharp b\}$, and $\{1\sharp a, 2\sharp b\} \oplus \{1\sharp b\} = \{2\sharp b\}$. The operator $\hookleftarrow$ will be called the *labeled replacement*, the operator $\oplus$ is an auxiliary operator that will be used to define *concatenation* and *weak concatenation* for the elements of $\mathcal{U}(\mathcal{L}, \langle\Delta^*\rangle)$.

The elements of $\mathcal{U}(\mathcal{L}, \langle\Delta^*\rangle)$ will be called *uniquely labeled sets of step-sequences*, and the elements of $\mathcal{L} \times \langle\Delta^*\rangle$ *labeled step-sequences*. In particular $\emptyset$ and $\alpha\sharp\varepsilon$ 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:

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

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

We also extend the operator $\in$ in the following way:

$$\begin{aligned} &\forall\alpha \in \mathcal{L}.\, \forall\tau \in \mathcal{U}(\mathcal{L}, \langle\Delta^*\rangle).\ \alpha \in \tau \ \Leftrightarrow\ \exists t \in \langle\Delta^*\rangle.\, \alpha\sharp t \in \tau. \\ &\forall a \in \Delta.\, \forall\tau \in \mathcal{U}(\mathcal{L}, \langle\Delta^*\rangle).\ a \in \tau \ \Leftrightarrow\ \exists\alpha \in \mathcal{L}.\, \exists t \in \langle\Delta^*\rangle.\, \alpha\sharp t \in \tau \wedge a \in t. \end{aligned}$$

For instance, $2 \in \{1\sharp a.a, 2\sharp a.\langle b.a\rangle\}$, but $3 \notin \{1\sharp a.a, 2\sharp a.\langle b.a\rangle\}$, $b \in \{1\sharp a.a, 2\sharp a.\langle b.a\rangle\}$, but $c \notin \{1\sharp a.a, 2\sharp a.\langle b.a\rangle\}$.
In summary, the following operators have been defined for the uniquely labeled sets of step-traces: *projection, selection, labeled replacement, concatenation* and *weak concatenation*, and the operator "in" ($\in$) has been extended.

14.2 Multi-Objects Trace Assertion Specifications

Let $TA = (\Sigma, \mathcal{O}, \Delta, \mathcal{C}, \delta, t_0)$ 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:

$$\mathcal{L}\sharp TA = (\mathcal{L} \times \Sigma, \mathcal{O}, \mathcal{L} \times \Delta, \mathcal{U}(\mathcal{L}, \mathcal{C}), \delta_{\mathcal{L}}, \emptyset)$$

where: $\mathcal{L} \times \Sigma$ is the set of *labeled actions*, $\mathcal{O}$ is the set of objects, $\mathcal{L} \times \Delta$ the set of *labeled actions-responses*, $\mathcal{U}(\mathcal{L},\mathcal{C})$ is the *uniquely labeled set of canonical step-traces*, $\delta_{\mathcal{L}}$ is the transition (extension) function defined as follows $\delta_{\mathcal{L}} : \mathcal{U}(\mathcal{L},\mathcal{C}) \times (\mathcal{L}\times\Delta) \to 2^{\mathcal{U}(\mathcal{L},\mathcal{C})}$, and:

$$\forall \tau \in \mathcal{U}(\mathcal{L},\mathcal{C}).\forall \alpha\sharp a{:}d \in \mathcal{L}\times\Delta.\ \ \delta_{\mathcal{L}}(\tau, \alpha\sharp a{:}d) = \begin{cases} \{\ \tau \hookleftarrow \{\alpha\sharp t\}\ \mid\ t \in \delta(\tau|\alpha, a{:}d)\ \} & \text{if } \delta(\tau|\alpha, a{:}d) \neq \emptyset \\ \emptyset & \text{if } \delta(\tau|\alpha, a{:}d) = \emptyset \end{cases}$$

The above definition assumes *self-initialization* of modules, i.e. the first (normal, not misuse) system call initializes a given object in the module. The pair $(\mathcal{L}, TA)$ describes $\mathcal{L}\sharp TA$ completely, since $\mathcal{L}\sharp TA$ 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 5 (without enhancement) describe completely the self-initializing multi-stack module. We may easily derive that in such a case (we specify normal behaviour only so far): $\delta_{\mathcal{L}}(\emptyset, st1\sharp PUSH(3)) = \{st1\sharp PUSH(3)\}$, while $\delta_{\mathcal{L}}(\emptyset, st1\sharp POP) = \emptyset$. If $\tau = \{st1\sharp PUSH(3).PUSH(1), st2\sharp\varepsilon, st3\sharp PUSH(5)\}$, then we have $\delta_{\mathcal{L}}(\tau, st1\sharp POP) = \{st1\sharp PUSH(3), st2\sharp\varepsilon, st3\sharp PUSH(5)\}$, while $\delta_{\mathcal{L}}(\tau, st2\sharp POP) = \emptyset$. In the sequel, except theory part, we shall prefer to write $PUSH(st1, 3).PUSH(st1, 1)$ instead of $st1\sharp PUSH(3).PUSH(1)$.

The *normal behaviour* described by $\mathcal{L}\sharp TA$ is given by:

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

where $\delta^*_{\mathcal{L}}$ is a standard extension of $\delta_{\mathcal{L}}$ onto $\mathcal{U}(\mathcal{L},\mathcal{C}) \times (\mathcal{L}\times\Delta)^* \to 2^{\mathcal{U}(\mathcal{L},\mathcal{C})}$ (see Section 6.1).

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

For every $\mathcal{L}\sharp TA$, let $\delta^{\emptyset}_{\mathcal{L}}$ denote the following function:

$$\forall \tau \in \mathcal{U}(\mathcal{L},\mathcal{C}).\ \forall \alpha\sharp a{:}d \in \mathcal{L}\times\Delta.\ \ \delta^{\emptyset}_{\mathcal{L}}(\tau, \alpha\sharp a{:}d) = \begin{cases} \delta_{\mathcal{L}}(\tau, \alpha\sharp a{:}d) & \text{if } \alpha \in \tau \\ \emptyset & \text{if } \alpha \notin \tau \end{cases}$$

A *multi-object trace assertion specification* we mean a tuple:

$$MTA = (\mathcal{L}\sharp TA, \Sigma_{glob}, \mathcal{O}_{glob}, \Delta_{glob}, \delta_{glob})$$

where $\mathcal{L}\sharp TA$ as above, Σ_{glob} is the set of *global actions*, $\mathcal{O}_{glob}$ is the set of *global outputs*, Δ_{glob} is the set of *global action-response events*, δ_{glob} the *global transition* (extension) function, and:

$$\Delta_{glob} \subseteq (\bigcup_{i=0}^{\infty}(\Sigma_{glob}\times\mathcal{L}^i) \times \mathcal{O}_{glob}) \cup (\bigcup_{i=0}^{\infty}(\Sigma_{glob}\times\mathcal{L}^i)).$$
$$\delta_{glob} : \mathcal{U}(\mathcal{L},\mathcal{C}) \times \Delta_{glob} \to 2^{\mathcal{U}(\mathcal{L},\mathcal{C})}.$$

The trace assertion TA will be called a *kernel* of MTA.

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

For every MTA we define *the transition function* $\hat{\delta} : \mathcal{U}(\mathcal{L},\mathcal{C}) \times ((\mathcal{L}\times\Delta) \cup \Delta_{glob}) \to 2^{\mathcal{U}(\mathcal{L},\mathcal{C})}$, where

$$\forall \tau \in \mathcal{U}(\mathcal{L},\mathcal{C}).\, \forall p \in (\mathcal{L}\times\Delta) \cup \Delta_{glob}.\ \hat{\delta}(\tau,p) = \begin{cases} \delta_{\mathcal{L}}(\tau,p) & \text{if } p \in \mathcal{L}\times\Delta \wedge new \notin \Sigma_{glob} \\ \delta^{\emptyset}_{\mathcal{L}}(\tau,p) & \text{if } p \in \mathcal{L}\times\Delta \wedge new \in \Sigma_{glob} \\ \delta_{glob}(\tau,p) & \text{if } p \in \Delta_{glob} \end{cases}$$

The system call $new(\alpha) \in \Sigma_{glob} \times \mathcal{L}$ is defined as follows:

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

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

The global system call $new(\alpha)$ which just creates a new instance of the TA, is usually accompanied by the system call $kill(\alpha) \in \Sigma_{glob} \times \mathcal{L}$ (name may vary), which kills an instance of TA labeled α. The $kill(\alpha)$ is defined as follows:

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

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

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

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

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 specifcation, Mealy form* and *state constructors* can easily be introduced for multi-objects modules. We left their formal definitions for a reader. The counterparts of Lemma 10.1, Proposition 10.2 and Proposition 12.1 also do hold for multi-objects trace specifications. Figure 10 represents a full self-initialized multi-object trace assertion speciffication for the Cross module that was introduced and analysed in [17]. The specification of the Cross module caused some problems when the older convention and techniques were used [17]. The Cross specification is non-deterministic and output independent. The module implements up to two sets, labeld 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 guven 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 11 represents the full multi-objects trace assertion specification of Multi-Stack with *new*, *concatenate*, and *kill* as global systems calls. The specification is deterministic, output-independent and non self-initializing. For multi-objects modules the specification format is extended.

14.3 Format for Multi-Objects Trace Assertion Specification

A Multi-Objects Trace Assertion Specification in the standard form consists of six, seven or eight sections: *Labels, Syntax, Local Canonical Step-Traces, Local Trace Assertions, Global Canonical Step-Traces, Global Trace Assertions and Extended Local Trace Assertions*, and *Dictionary*. The *Global Canonical Step-Traces* are always redundant and may be omitted. The *Extended Local Trace Assertions* may be partially or entirely redundant as well.

The *Labels* section specifies all the names that are allowed as labels. The *Syntax* section is expanded by the columns *Type* and *Code*. The *Type* may be *global*, if an action belongs to Σ_{glob} or *local*, if it belongs to Σ. *Codes* are used to make specification of canonical-step traces shorter (and easier to understand). Instead of use the full action-responce form we may use just a code, and for instance write i instead of $INSERT(i)$. The convention $stack_1 \sharp PUSH(i)$ and $new(stack_2)$ (or $(new, stack_2)$) works fine for the theory part, but for applications $PUSH(stack_1, i)$ and $new(stack_2)$ seems to be more natural. In the *Action-response Forms* column, we shall use "$*$" to indicate where the label variables and constants occur. The *Local Canonical Step-traces* section defines the canonical step-traces (usually using *codes* from Syntax section) of the *kernel*, and the *Local Trace Assertions* section defines the trace assertions (mapping δ) of the *kernel*. The rules for writing trace assertions are exactly the same as described in Section 13 (Specification Format). The section *Global Canonical Step-traces* is always redundant, but its inclusion may increase the readability. The convention

$$\{E(x_i)\}_{i=j}^{k}$$

as a shorthand for $\{E(x_j), E(x_{j+1}), ..., E(x_k)\}$ and $\emptyset$ if $k < j$, may be used. The rules for *Global Trace Assertions*, that define the mapping δ_{glob} and *Extended Local Trace Assertions* are the same as for *Local Trace Assertions*, that defines $\delta_{\mathcal{L}}$ or $\delta_{\mathcal{L}}^{\emptyset}$. The *Extended Local Trace Assertions* may be in part or entirely redundant. In order to keep the number of symbols used as small as possible, we shall use the symbol $\hat{\delta}$ for both *Global Trace Assertions* and *Extended Local Trace Assertions*. The symbol % will denote the enhancement part of the specification.

15 Trace Assertion Method and Algebraic Specification

There are strong similarities between the Trace Assertion Method and the *Algebraic Specification Method* (see [9, 37]) which is one of the most well-known approaches to specifying abstract data types. Examples of similarities:

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 [9]).

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 action-response pairs of visible functions) correspond to auxiliary/hidden functions in algebraic specification.

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

Syntax of Access Programs

Name	Type	Argument	Value	Action-response Forms	Code
$INSERT$	local	0 or 1		$INSERT(*, i)$:nil	i
$TEST$	local	0 or 1	*Boolean*	$TEST(*, i)$:d	
$CROSS$	global			$CROSS$:nil	

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

Local Trace Assertions

$\delta(t, INSERT(*, i)) =$

Equivalence
$t \frown i$

$\delta(t, TEST(*, i)$:$d) =$

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

Global Canonical Step-traces (Redundant)

$$global\_canonical(\tau) \Leftrightarrow \tau = \emptyset \vee ((\tau = \{a\sharp t\} \vee \tau = \{b\sharp t\}) \wedge canonical(t)) \vee (\tau = \{a\sharp t_1, b\sharp t_2\} \wedge canonical(t_1) \wedge canonical(t_2))$$

$\tau_0 = \emptyset$

Global Trace Assertions
$\hat{\delta}(\tau, CROSS) =$

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

Extended Local Trace Assertions (Redundant, except enhancements)

$\hat{\delta}(\tau, INSERT(\alpha, i)) =$

Equivalence
$\tau \frown \{\alpha\sharp i\}$

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

Condition	Equivalence
$\alpha\sharp i \in \tau \wedge d = true$	τ
$\alpha \in \tau \wedge \alpha\sharp i \notin \tau \wedge d = false$	τ
% $\alpha \notin \tau$	τ

Figure 11: Full Trace Assertion Specification for (a self-initializing) Cross Module

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

Syntax of Access Programs

Name	Type	Argument	Value	Action-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	$3 \times label$		$concatenate(*,*,*)$	
$kill$	global	*label*		$kill(*)$	

Local Canonical Step-traces

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

$t_0 = \varepsilon$

Local Trace Assertions

$\delta(t, POP(*)) =$

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

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

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

$\delta(t, TOP(*)$:$d) =$

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

Global Canonical Traces (Redundant)

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

$\tau_0 = \emptyset$

Global Trace Assertions

$\hat{\delta}(\tau, new(\alpha)) =$

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

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

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

$\hat{\delta}(\tau, kill(\alpha)) =$

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

Dictionary

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

Figure 12: Full Trace Assertion Specification for Multiple Stack Module. Extended Local Trace Assertions are omitted as redundant.

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 [9]:

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

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 [9]):

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

In the second, explicit, case we may replace "=" by "$\stackrel{df}{=}$", 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 the vast majority of peoples seem to consider *explicit* definitions as 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-explained. For more complex modules, as for example parts of protocols [7, 15], parts of software for aircraft control [34], or intra-processor, inter-process communication via mailboxes [35] both defining and understanding implicit equations might be difficult (how simple equational definitions of Unique Integer or Cross module would look like?).

The second difference is

2. The underlying models for algebraic specification are *abstract algebras* [5], 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 specification provides syntactic facilities which makes it possible in certain cases to specify a function by a single trace assertion, while 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 [9]) are necessary in algebraic specification

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. 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 Fig. 11), and

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

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

by

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

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

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

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

where $PUSH : Stack \times Integers \rightarrow Stack \times Integers$. 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* ([9, 37]), 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 for some special cases. Those transformations will also emphasize similarities and differences.

Let FTA be a deterministic and output-independent *full* trace assertion specification, and let $FTA^{Mealy} = (\Sigma, \mathcal{O}, \pi(\mathcal{C}), \delta_\Sigma, v, t_0)$, be its Mealy form. Let $\Sigma_\delta \subseteq \Sigma$ be defined as follows:

$$a \in \Sigma_\delta \quad \Leftrightarrow \quad \forall t \in \pi(\mathcal{C}).\, v(t,a) = nil,$$

and let $\Sigma_v = \Sigma \setminus \Sigma_\delta$. Assume that

$$\forall a \in \Sigma_v. \forall t \in \pi(\mathcal{C}).\ \delta_\Sigma(t,a) = \{t\}.$$

For example the Stack Module from Figure 6 has the above property with $\Sigma_v = \{TOP\}$ and $\Sigma_\delta = \{POP\} \cup \{PUSH(i) \mid\ i \text{ is an integer}\}$.

For every $x \in \langle \Sigma^* \rangle$, let $\tilde{x} \in \langle \Sigma_\delta^* \rangle$ be a string derived from x by erasing all elements of Σ_v from x. For instance if $x = PUSH(1).TOP.PUSH(3)$ then $\tilde{x} = PUSH(1).PUSH(3)$. If $\Sigma_v = \{a, d\}$, $\Sigma_\delta = \{b, c\}$, and $x = b.\langle a.b.c \rangle.\langle a.d \rangle.c$ then $\tilde{x} = b.\langle b.c \rangle.c$

Since FTA is deterministic, then for all $t \in \pi(C)$, $x \in \langle \Sigma^* \rangle$,

$$\delta_\Sigma^*(t,x) = \delta_\Sigma^*(t,\tilde{x})$$

and we may replace $t \in \pi(\mathcal{C})$ by $\tilde{x} \in \widetilde{\pi(\mathcal{C})}$. Hence without loss of generality we may assume that $\pi(C) \subseteq \langle \Sigma_\delta^* \rangle$ (otherwise replace $\pi(\mathcal{C})$ by $\widetilde{\pi(\mathcal{C})}$ and modify δ_Σ respectively).

Let $\mathcal{O}_v = v(\pi(\mathcal{C})) \subseteq \mathcal{O}$, and let us consider the following two-sorts function algebra:

$$\mathcal{A}_{FTA} = (\pi(\mathcal{C}), \mathcal{O}_v; \Sigma_\delta^{\mathcal{A}} \cup \Sigma_v^{\mathcal{A}} \cup \{t_0\}),$$

where $\pi(\mathcal{C})$ and $\mathcal{O}_v$ are domains, $\Sigma_\delta^{\mathcal{A}} \cup \Sigma_v^{\mathcal{A}} \cup \{t_0\}$ are functions with t_0 as the only constant. The elements of $\Sigma_\delta^{\mathcal{A}}$ are functions from $\pi(\mathcal{C})$ into $\pi(\mathcal{C})$, and the elements of $\Sigma_v^{\mathcal{A}}$ are functions from $\pi(\mathcal{C})$ into $\mathcal{O}$) defined as follows:

$$\begin{array}{lll} \hat{a} \in \Sigma_\delta^{\mathcal{A}} & \text{iff} \quad a \in \Sigma_\delta & \text{and} \quad \hat{a}(t) = s \Leftrightarrow \delta(t,a) = \{s\}, \\ \hat{a} \in \Sigma_v^{\mathcal{A}} & \text{iff} \quad a \in \Sigma_v & \text{and} \quad \hat{a}(t) = d \Leftrightarrow v(t,a) = d. \end{array}$$

For every $x \in \Sigma_\delta^*$, let $\hat{x} : \pi(\mathcal{C}) \to \pi(\mathcal{C})$ be defined as follows: if $x = \varepsilon$ then $\hat{x}(t) = t$, if $x = a_1. \ldots .a_k$ then $\hat{x}(t) = \hat{a_k}(...(\hat{a_1}(t))...)$.

$\mathcal{A}_{FTA}$ and FTA^{Mealy} are *equivalent* in the sense that:

$$\delta_\Sigma^*(t,x) = \{\hat{\tilde{x}}(t)\} \quad \text{and} \quad v(t,a) = \hat{a}(t) \text{ if } a \in \Sigma_v.$$

In the same sense every $t \in \pi(\mathcal{C})$ can be identified as a *canonical ground term* [9]. For example $PUSH(1).PUSH(2)$ corresponds to $PUSH(PUSH(\varepsilon,1),2)$. Hence the algebra $\mathcal{A}_{FTA}$ can be seen as a *canonical term algebra* that is equivalent to FTA.

Consider the following many-sorted algebra

$$\mathcal{A} = (S_0, S_1, ..., S_k; Op)$$

where S_i, $i = 0, ...k$, are *domains*, and Op is the set of *operations* (*functions*). Assume that $Op \cap S_0 = \{s_0\}$, i.e. there is only one constant with a value in S_0. Assume also that every operation $\hat{a} \in Op$ which is not a constant, is either of the form $\hat{a} : S_0 \to S_{i_a}$, or

$$\hat{a} : S_0 \times \underbrace{S_{i_1} \times ... \times S_{i_{k_a}}}_{i_j \neq 0} \to S_{i_a},$$

where $i_a = 0, ..., k$. For every $\hat{a} \in Op$, we write $range(\hat{a}) = S_{i_a}$. The domain S_0 is called *domestic*, the others are *foreign*, and the algebra is called with *one domestic domain*. Most algebraic specifications in the literature result in the algebras with one domestic domain.

If $\hat{a} : S_0 \times S_{i_1} \times ... \times S_{i_{k_a}} \to S_{i_a}$, then let $fd(\hat{a}) = S_{i_1} \times ... \times S_{i_{k_a}}$ ($f$oreign $d$omains of $\hat{a}$), if $\hat{a} : S_0 \to S_{i_0}$ then $fd(\hat{a}) = \emptyset$.

For every operation $\hat{a}$, let a denotes the name of this operation. Define:

$$\begin{array}{l} \Sigma = \{(a,r) \mid \hat{a} \in Op \wedge r \in fd(\hat{a})\} \cup \{a \mid \hat{a} \in Op \wedge fd(\hat{a}) = \emptyset\}, \\ \mathcal{O} = \bigcup_{\hat{a} \in Op} range(\hat{a}) \setminus S_0 \cup \{nil\}. \end{array}$$

We shall write rather $a(r) \in \Sigma$ instead of $(a,r) \in \Sigma$. Let $\Sigma \subseteq \Sigma$ be defined as follows:

$$\Sigma_\delta = \{a \mid a \in \Sigma \wedge range(\hat{a}) = S_0\} \cup \{a(r) \mid a(r) \in \Sigma \wedge range(\hat{a}) = S_0\}$$

and let $\Sigma_v = \Sigma \setminus \Sigma_\delta$.

We define $\delta_\Sigma : S_0 \times \Sigma \to S_0$, $v : S_0 \times \Sigma \to \mathcal{L}$ as

1. if $range(\hat{a}) = S_0$ and $fd(\hat{a}) = \emptyset$ then $\delta_\Sigma(t,a) = \{s\} \wedge v(t,a) = nil \;\Leftrightarrow\; \hat{a}(t) = s$,
2. if $range(\hat{a}) = S_0$ and $fd(\hat{a}) \neq \emptyset$ then $\delta_\Sigma(t,a(r)) = \{s\} \wedge v(t,a) = nil \;\Leftrightarrow\; \hat{a}(t,r) = s$,
3. if $range(\hat{a}) \neq S_0$ and $fd(\hat{a}) = \emptyset$ then $\delta_\Sigma(t,a) = \{t\} \wedge v(t,a) = d \;\Leftrightarrow\; \hat{a}(t) = d$,
4. if $range(\hat{a}) \neq S_0$ and $fd(\hat{a}) \neq \emptyset$ then $\delta_\Sigma(t,a(r)) = \{t\} \wedge v(t,a) = d \;\Leftrightarrow\; \hat{a}(t,r) = d$,

Without any loss of generality we may assume that S_0 is the set of *canonical ground terms* [9, 37], so every element of S_0 can easily be interpreted as a canonical trace ($PUSH(PUSH(\varepsilon,1),2)$ corresponds to $PUSH(1).PUSH(2)$).

Let us define

$$FTA_{\mathcal{A}}^{Mealy} = (\Sigma, \mathcal{O}, S_0, \delta_\Sigma, v, s_0).$$

One may verify that $FTA_{\mathcal{A}}^{Mealy}$ is a correctly defined Full Trace Assertion Specification in Mealy form. To show an equivalence of $\mathcal{A}$ and $FTM_{Mealy}^{\mathcal{A}}$ we proceed in the same way as for the equivalence of $\mathcal{A}_{FTA}$ and FTA_{Mealy}. For every $x \in \Sigma_\delta^*$ we define $\hat{x} : S_0 \to S_0$, as follows. If $x = \varepsilon$ then $\hat{x}(t) = t$, if $x = y.a$ then $\hat{x}(t) = \hat{a}(\hat{y}(t))$, if $x = y.a(r)$ then $\hat{x}(t) = \hat{a}(\hat{y}(t), r)$. For every $x \in \Sigma$, $\tilde{x} \in \Sigma^*\delta$ is the string derived from x by erasing all elements of Σ_v from x.

The algebra $\mathcal{A}$ and the trace assertion specification $FTA_{\mathcal{A}}^{Mealy}$ are *equivalent* in the sense that:

$$\delta_\Sigma^*(t,x) = \{\hat{\tilde{x}}(t)\} \quad \text{and} \quad \begin{array}{l} v(t,a) = \hat{a}(t) \text{ if } a \in \Sigma_v \wedge fd(\hat{a}) = \emptyset, \text{ or} \\ v(t,a(r)) = \hat{a}(t,r) \text{ if } a \in \Sigma_v \wedge fd(\hat{a}) \neq \emptyset. \end{array}$$

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 [37]). The trace assertion method is better suited for specifying complex interface modules as for instance communication protocols [7, 15, 34, 35]. 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 [13] 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.

16 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 action-response events,
- the structure of the trace assertion specification is entirely described on the bases of normal behaviour only,

- exceptional behaviour is specified separately as an enhancement of normal behaviour, and such an enhancement may be added to the trace assertion specification,
- 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,
- the determinism is defined and handled in the way it is done in automata theory and is differentiated from the concept of output independence,
- 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 [13, 32, 36] nor non-sequential modules are considered in this paper. For non-sequential models a possible delay between an action and its response must be modeled, so "true-concurrency" models should rather be used [22]. 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

The author would like to thank D. Parnas, A. J. van Schouwen, and K. Stencel, for many useful comments and (e-mail) discussions.

References

[1] J.-R. Abrial, *The B-Book*, Cambridge University Press, 1996.

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

[3] W. Bartussek, D.L. Parnas, Using Assertions About Traces to Write Abstract Specifications for Software Modules, Proc. 2nd Conf. on European Cooperation in Informatics, *Lecture Notes in Computer Science 65*, Springer 1978, pp. 211-236.

[4] A. Blikle, An analysis of programs by algebraic means, In: A. Mazurkiewicz, Z. Pawlak (eds.), *Mathematical Foundations of Computer Science*, Banach Centre Publications, Vol. II, Polish Scientific Publishers 1977, 167-214.

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

[6] D. Comer, T. Fossum, *Operating System Design. The XINU Approach*, Prentice Hall 1988.

[7] F. Courtois, D. L. Parnas, Formally Specifying a Communication protocol Using the Trace Assertion Method, CRL Report No. 269, McMaster University, Hamilton, Ontario, Canada 1993.

[8] V. Diekert, G. Rozenberg, (eds.), *The Book of Traces*, World Scientific, 1995.

[9] H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification 1*, Springer-Verlag, 1985.

[10] S. Eilenberg, *Automata, Languages and Machines*, vol A, Academic Press, 1974.

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

[12] S. J. Garland, D. C. Luckham, Program Schemes, Recursion Schemes and Formal Languages, *J. Comp. System Sci.* 7 (1973), 119-160.

[13] C. L. Heitmayer, R. D. Jeffords, B. G. Labaw, Automated Consistency Checking of Requirements Specification, *ACM Trans. on Software Eng. and Methodologies*, 5, 3 (1996), 231-261.

[14] C. A. R. Hoare, A Model for Communication Sequential Processes, In: R.M. McKeag, A.A. Macnaghten (eds.), *On the Construction of Programs*, Cambridge University Press, 1980.

[15] D.M. Hoffman, The Trace Specification of Communication Protocols, *IEEE Transactions on Computers 34*, 12 (1985), 1102-1113.

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

[17] M. Iglewski, M. Kubica, J. Madey, Trace Specification of Non-deterministic Multi-object Modules, TR 95-05(205), Institute of Informatics, Warsaw University, Warsaw, Poland, 1995.

[18] M. Iglewski, J. Mincer-Daszkiewicz, J. Stencel, Some Experiences with Specification of Non-deterministic Modules, RR 94/09-7, Université du Québec à Hull, Hull, Canada, 1994.

[19] M. Iglewski, J. Madey, K. Stencel, On Fundamentals of the Trace Assertion Method, RR 94/09-6, Université du Québec à Hull, Hull, Canada,

[20] R. Janicki, Analysis of coroutines by means of vectors of coroutines, *Fundamenta Informaticae* 2, 3 (1979), 289-316.

[21] R. Janicki, Towards a Formal Semantics of Parnas Tables, *Proc. 17th International Conference on Software Engineering*, Seattle, Washington, USA, 1995, pp. 231-240.

[22] R. Janicki, M. Koutny, Structure of Concurrency, *Theoretical Computer Science* 112 (1993), 5-52.

[23] R. Janicki, D. L. Parnas, J. Zucker, Tabular Representations in Relational Documents, in C. Brink, G. Schmidt (Eds.): *Relational Methods in Computer Science*, Springer-Verlag, 1997.

[24] A. Mazurkiewicz, Iteratively computable relations, *Bull. Acad. Polon. Sci., Sér. Sci. Math. astronom. Phys.* 20 (1972), 793-795.

[25] J. McLean, A Formal Foundations for the Abstract Specification of Software, *Journal of the ACM*, 31,3 (1984), 600-627.

[26] H. D. Mills, Stepwise Refinement and Verification in Box-Structure Systems, *Computer* (1988), 23-26.

[27] T. Norvell, On Trace Specifications, CRL Report 305, TRIO, McMaster University, Hamilton, Canada, 1995.

[28] D. Parnas, A Technique for Software Module Specification with Examples, *Comm. of ACM*, 15,5 (1972), 330-336.

[29] D. Parnas, Personal communication, 1995.

[30] D. Parnas, J. Madey, M. Iglewski, Precise Documentation of Well-Structured Programs, *IEEE Transactions on Software Engineering*, 20, 12 (1994), 948-976.

[31] S. J. Prowell, Sequence-Based Software Specification, Ph.D. Thesis, University of Tennessee, Knoxville, Tennessee, U.S.A., 1996.

[32] D. Parnas, Y. Wang, The Trace Assertion Method of Module Interface Specification, TR 89-261, CIS, Queen's University, Kingston, Canada, 1989.

[33] G. Rozenberg, R. Varraedt, Subset languages of Petri nets, *Theoretical Computer Science* 26 (1983) 301-323.

[34] A. J. van Schouwen, The A-7 requirements model: Re-examination for real time systems and an application to monitoring systems, TR-90-276, Queen's University, Kingston, Ontario, Canada.

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

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

[37] M. Wirsing, Algebraic Specification, in J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science*, Vol 2., Elsevier Science Publ., 1990, pp. 675-788.