Theory of Deterministic Trace-Assertion Specifications*

Technical Report CS–2004–30, School of Computer Science, University of Waterloo, Waterloo, ON, Canada N2L 3G1

Janusz Brzozowski[1] and Helmut Jürgensen[2]

[1] School of Computer Science, University of Waterloo, Waterloo, ON, Canada N2L 3G1
brzozo@uwaterloo.ca http://maveric.uwaterloo.ca
[2] Department of Computer Science, The University of Western Ontario, London, ON, Canada N6A 5B7
and
Institut für Informatik, Universität Potsdam, August-Bebel-Str. 89, 14482 Potsdam, Germany
helmut@uwo.ca http://www.csd.uwo.ca/faculty_helmut.htm

Abstract. A software module is an abstraction of a program: it has a well defined functionality, operations by which the environment accesses the program, and outputs. Traces are sequences of operations. Trace assertions constitute a particular formalism for abstract specification of software modules. Certain traces are identified as *canonical,* and all traces are grouped in equivalence classes, each of which is represented by a unique canonical trace. *Trace equivalence* captures the observational indistinguishability of traces. A *rewriting system* transforms any trace to its canonical equivalent.

A module can often be conveniently described by a (finite or infinite) automaton. In this paper we consider only deterministic connected automata. We show that any such automaton can be specified by canonical traces and trace equivalence; conversely, any set of canonical traces together with a trace equivalence uniquely defines an automaton.

For each state of an automaton, we select an arbitrary trace leading to that state as its canonical trace. Constructing trace equivalence amounts to finding a set of generators for state-equivalence, where two traces are state-equivalent if they lead to the same state. We present a simple algorithm for finding such a set of generators. Directly from these generators, we derive a rewriting system which yields a deterministic algorithm for transforming any trace to its canonical representative. This system is always confluent, and it is Noetherian (has only finite derivations) if and only if the set of canonical traces is prefix-continuous (a set is prefix-continuous if whenever a word w and a prefix u of w are in the set, then all the prefixes of w longer than u are also in the set). Each prefix-continuous set corresponds to a spanning forest of the automaton and *vice versa*.

We apply our algorithms to specify several commonly used modules.

* This report was posted in May 2004, and revised in August 2004. Part of the material from the May version has appeared in [5].

1 Introduction

Formal methods are still not universally accepted in the design of commercial software. However, "scenarios" or "use cases" have become quite popular; for example, see [21]. A scenario usually consists of an informal description of a sequence of events specifying a part of the behavior of a software module. For example, a parking machine might satisfy the following scenario: "If a dollar is inserted in the coin slot and then the button is pushed, the machine issues a receipt valid for one hour."

Many software modules can be conveniently specified by automata. The *trace-assertion method* is somewhat similar in nature to the scenario approach, and introduces automata rather indirectly. Thus, instead of defining the input and output alphabets, state set, transition and output functions of an automaton, the trace-assertion method first identifies a set of important traces (sequences of operations), called "canonical," and then examines the remaining traces and declares them to be equivalent to the appropriate canonical traces. Canonical traces are analogous to scenarios in that they provide a partial description of the automaton. The equivalences supply the missing transitions. As a separate issue, outputs are added later. The proponents of trace assertions believe that this approach is more abstract than a specification by an automaton in which states are explicitly identified.

Our work is motivated by a series of papers written by D. L. Parnas and his collaborators, and other authors, over the past 25 years. The trace-assertion method for specifying software modules was introduced in 1977 by Bartussek and Parnas [1] (reprinted as [3] in 2001); a slightly modified version of [1] appeared as [2] in 1978. The method has undergone several changes since the original paper: see, for instance, [7–9, 11, 14, 17, 19, 20] for more details and additional references. The main inspiration for our work is the 1994 paper by Wang and Parnas [20].

We provide a theory of deterministic trace-assertion specifications. Trace assertions are abstract specifications of modules. For such specifications, it is assumed that modules are representable by (finite or infinite) automata. Trace assertions then serve as "black-box" models of the automata. In fact, a trace-assertion specification is a particular way of defining an automaton. A complete trace-assertion specification consists of six parts: syntax, canonical traces, trace equivalence, legality, values, and a rewriting system. Traces are sequences of operations (function calls) of the module. The syntax part defines the domains and co-domains of the functions; a canonical trace is a representative of the set of all traces leading to the same state of the module; equivalence identifies the traces leading to the same state; legality distinguishes normal from abnormal sequences of calls; the "values" part defines the output values produced by certain function calls; the rewriting system allows us to transform any trace to its canonical form algorithmically.

In terms of automaton theory, traces are input words. From now on we use "trace" and "word" interchangeably. However, whenever the focus is on language-theoretic issues, we use "word," while we use "trace" to emphasize the

focus on applications. Every state is represented by a canonical trace leading to that state, and trace equivalence defines the transitions of the automaton. We do not restrict the automaton model to be finite, because no advantage is gained by doing so. Our theory is first developed in terms of semiautomata (automata without final states and without outputs), because trace equivalence can be handled conveniently in these more general structures. To obtain the complete trace-assertion specification we add outputs later.

Any trace leading to a state can be chosen as the canonical trace of that state. We show that constructing trace equivalence amounts to finding a set of generators for state-equivalence, where two traces are state-equivalent if they lead to the same state. We describe a simple algorithm for constructing a set of generators.

An argument sometimes raised against automata as specifications is that the representation of states necessarily implies a kind of implementation, whereas a description by traces is more abstract. We prove that, given a set of canonical traces and a trace equivalence, one can reconstruct the original automaton uniquely up to isomorphism. In this sense, trace-assertion specifications are no more abstract than specifications by automata.

To transform any trace to its canonical form algorithmically, we define a simple rewriting system directly from the generators of the trace equivalence, and prove that this system is always confluent. In general, the rewriting system may have infinite derivations. To remedy this, we impose a condition on the set of canonical traces. A set of words is prefix-continuous if whenever a word w and a prefix u of w are in the set, then all the prefixes of w longer than u are also in the set. Prefix-continuous sets include prefix-closed sets (where every word in the set has all of its prefixes in the set) and prefix codes (where no word in the set is a prefix of any other word in the set) as special cases. We prove that the rewriting system is Noetherian (has only finite derivations) if and only if the set of canonical traces is prefix-continuous.

Our method constitutes an algorithm for deriving a complete set of trace assertions directly from an automaton. Its simplicity and applicability is demonstrated on several common modules, such as stacks, queues, linked lists and sets.

Details of our mathematical results concerning the representation of semiautomata by canonical traces and equivalences can be found in [5]. In this paper we focus on the application of our theory to (software or hardware) module specifications. Consequently, some technical proofs are omitted here.

The remainder of the paper is structured as follows. We give a brief survey of previous work on trace-assertion specifications in Section 2. Section 3 introduces our terminology and notation. Arbitrary sets of canonical traces are studied in Section 4. Prefix-continuous sets of canonical traces are discussed in Section 5. Our theory is illustrated in Section 6 with the simple example of a counter. A more complete and more complex example, that of a stack, is given in Section 7, where the "values" part of the specification is introduced to handle outputs. In Section 8 we discuss the set module, which shows that the trace-

assertion method can be awkward in some applications. A bounded stack is treated in Section 9, and Section 10 concludes the paper. Four somewhat more challenging examples are presented in the appendices.

2 Background

The explicit goal of [1] was to make the specification of software modules independent of implementations, that is, to abstract from implementation and operational issues. In [1], Bartussek and Parnas use the concepts of syntax, legality, equivalence, and values. Canonical traces are not used, but the concept appears implicitly. It was noted there that it would be important for the formal verification of module correctness that equivalence and legality be recursive. Using the approach proposed in [1] and several subsequent papers [2, 14, 17], it is awkward to prove some equivalences, because the definitions of equivalence and legality depend on each other, and the definition of equivalence, if used directly, involves an infinite test.

In 1984, McLean provided a model-theoretic framework for the trace specification method [14]. It is based on first-order logic with equality, and with equivalence and legality defined as special predicates. Soundness and completeness (in the sense of logic) are proved: any statement about traces which has a formal proof is semantically true, and every semantically true statement has a formal proof. The definitions of equivalence and legality still depend on each other, and equivalence is still defined using an infinite test. It is assumed that the empty trace is legal, the prefix of a legal trace is legal, and only legal traces can return values.

In a 1992 paper [15], McLean retains the definitions of equivalence and legality mentioned above, but points out that the definition of equivalence implies that equivalence is a right congruence. We show that the right-congruence property is sufficient, that is, the dependence of equivalence on legality is unnecessary.

The work in the 1994 paper by Wang and Parnas [20] is restricted to finite automata, whereas our methods apply to both finite and infinite automata. The interdependence of the definitions of equivalence and legality is removed in [20] (see also [17]). *Canonical traces* are identified as representatives of equivalence classes, and a *reduction function* transforms any trace to its canonical representative. Explicit reference is made to a state machine (deterministic and finite) representing the software module, and four assumptions, absent in earlier work, are introduced, namely: (1) the empty trace must be canonical; (2) equivalence must be a right congruence; (3) the reduction function, when applied to a canonical trace, returns that same trace; (4) reduction of a long trace can be performed by first reducing a prefix of the trace and then reducing the result with the remainder of the trace appended. No specific rule for the choice of canonical traces is given in [20] except assumption (1) above. We show that assumption (1) is not necessary and that properties (3) and (4) hold automatically.

To prove trace equivalence, [20] uses term (rather than string) rewriting systems. One applies term rewriting rules to a given trace to obtain the equivalent

canonical trace. This process is not necessarily convergent. A sufficient condition for convergence is that the rewriting system be *confluent* and *Noetherian.* A heuristic, called *smart rewriting*, which leads to the canonical trace in many, but not all, cases is used in [20].

Term rewriting introduces an unmanageable complexity into the problem; this can be avoided by string rewriting over an infinite, but recursively enumerable alphabet. In this paper we use only string rewriting, and refer to it simply as rewriting. We show that, if string rewriting is used, confluence always holds. Moreover, the rewriting system is Noetherian if and only if the set of canonical traces is prefix-continuous.

After [20], all publications on the trace-assertion method seem to rely on an unexplained choice of the canonical traces. Because the "natural" or "intuitive" choice often happens to lead to a provably confluent and Noetherian rewriting system, this approach usually works.

Basic ideas concerning formal descriptions of the trace-assertion method were presented in 1994 in [10]. That work deals with modules describable by nondeterministic Mealy automata. A concept similar to legality is used; it is called feasibility. Canonical traces are chosen arbitrarily, except that it is assumed that the empty trace is canonical. Two equivalences are used: "observational equivalence," $\overset{o}{\equiv}$, and "specification-equivalence," $\overset{s}{\equiv}$. It is noted that $\overset{s}{\equiv} \subseteq \overset{o}{\equiv}$. Non-canonical one-letter extensions (called single-event extensions in [10]) of canonical traces are declared equivalent to appropriate canonical traces. Using these constructs, the authors propose a Mealy automaton as a formal model of a trace-assertion specification. In general, this automaton is not reduced.

The work reported in [9] uses the ideas of [10]. It focusses mostly on the implementation of the trace-assertion method including its syntactic representation. It provides a comprehensive view of the field as of 1997. In the definition of equivalence, it follows [10], in that two equivalences are considered.

In [11], among other items, the problems of non-deterministic modules and their ramifications are investigated. We do not consider non-deterministic specifications in this paper.

The trace-assertion method has also been used for time-dependent systems like communication protocols [7]. Timing conditions were not a part of the original proposal in [1]. The work in [7, 8] proposes a heuristic for choosing canonical traces. We prove in this paper that this heuristic is indeed appropriate. In [16], trace-assertion methods are used to study security issues in softwares systems.

For a 1991 survey of formal specification methods for software modules see [19]; this survey was updated in 1994 [19]. A more recent comprehensive survey, published in 2002 by Madey [13], although focussed on a functional approach to software design, has an extensive bibliography on formal specifications, including trace-assertion methods.

3 Terminology and Notation

We denote by Z and P the sets of integers and nonnegative integers, respectively. Purely for convenience, we use integers as the data that is stored in the various modules we describe; there is no loss of generality in this assumption. If Σ is an alphabet (finite or infinite), then Σ^* and Σ^+ denote the set of all words and the set of all nonempty words, respectively, over Σ. The empty word is ϵ. For $w \in \Sigma^*$, $|w|$ denotes the length of w. If $w = uv$, for some $u, v \in \Sigma^*$, then u is a *prefix* of w. A set $X \subseteq \Sigma^*$ is a *prefix code* if no word of X is the prefix of any other word of X. Note that, with this definition, the set $\{\epsilon\}$ is a prefix code, in contrast to most of the commonly used definitions. A set X is *prefix-closed* if, for any $w \in X$, every prefix of w is also in X. A set X is *prefix-continuous* if, whenever $x = uav$ is in X, $a \in \Sigma$, then $u \in X$ implies $ua \in X$. Both prefix codes and prefix-closed sets are prefix-continuous.

3.1 Semiautomata and Equivalences

By a *deterministic initialized semiautomaton*, or simply *semiautomaton*, we mean a tuple $S = (\Sigma, Q, \delta, q_\epsilon)$, where Σ is a nonempty input alphabet, Q is a nonempty set of states, $\delta : Q \times \Sigma \to Q$ is the transition function, and $q_\epsilon \in Q$ is the initial state. In general, we do not assume that Σ and Q are finite. As usual, we extend the transition function to words by defining $\delta(q, \epsilon) = q$, for all $q \in Q$, and $\delta(q, wa) = \delta(\delta(q, w), a)$. A semiautomaton is *connected* if every state is reachable from the initial state. We consider only connected semiautomata. Thus, for every $q \in Q$, there exists $w \in \Sigma^*$ such that $\delta(q_\epsilon, w) = q$. For any $w \in \Sigma^*$, we define $q_w = \delta(q_\epsilon, w)$.

For a semiautomaton $S = (\Sigma, Q, \delta, q_\epsilon)$, the *state-equivalence* relation $\equiv_\delta$ on Σ^* is defined by

$$w \equiv_\delta w' \Leftrightarrow q_w = q_{w'}, \tag{1}$$

for $w, w' \in \Sigma^*$. Note that $\equiv_\delta$ is an equivalence relation, and also a right congruence, that is, for all $x \in \Sigma^*$,

$$w \equiv_\delta w' \Rightarrow wx \equiv_\delta w'x. \tag{2}$$

3.2 Automata

By a *deterministic automaton*, we mean a tuple $A = (\Sigma, Q, \delta, q_\epsilon, F)$, where $(\Sigma, Q, \delta, q_\epsilon)$ is a semiautomaton, and $F \subseteq Q$ is the set of final states. A word $w \in \Sigma^*$ is accepted by A if and only if $q_w \in F$. The language accepted by A is $L(A) = \{w \mid q_w \in F\}$.

By a *generalized Mealy automaton*, or simply *automaton*, we mean a deterministic automaton M with an output alphabet and an output function. More precisely, $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$, where $(\Sigma, Q, \delta, q_\epsilon, F)$ is a deterministic automaton, Ω is the output alphabet, and $\nu : Q \times \Sigma \to \Omega$ is a partial function called the output function. Note that a deterministic automaton is a generalized

Mealy automaton without outputs, and a generalized Mealy automaton is a normal Mealy automaton with accepting states. As before, $L(M) = \{w \mid q_w \in F\}$.

If f and g are partial functions, by $f(x) = g(y)$ we mean that either both values are undefined, or both are defined and they are equal. The partial function $\nu : Q \times \Sigma \to \Omega$ uniquely determines a partial function $\nu' : \Sigma^+ \to \Omega$ as follows: For $w \in \Sigma^*$ and $a \in \Sigma$, $\nu'(wa) = \nu(q_w, a)$. In the sequel, we refer to ν' simply as ν.

The *generalized Nerode equivalence* relation $\equiv_M$ on Σ^* is defined as follows: for $w, w' \in \Sigma^*$, $w \equiv_M w'$ if and only if

$$\forall u \in \Sigma^*, \forall a \in \Sigma, \quad wu \in L(M) \leftrightarrow w'u \in L(M) \ \wedge \ \nu(wua) = \nu(w'ua). \quad (3)$$

Note that the following always holds: $w \equiv_\delta w' \Rightarrow w \equiv_M w'$. An automaton M is *reduced* with respect to the equivalence $\equiv_M$ if and only if $w \equiv_M w' \Rightarrow w \equiv_\delta w'$. Thus, in a reduced automaton we always have $\equiv_M = \equiv_\delta$.

In some of the literature on trace assertions (for example, [9, 10]) the generalized Nerode equivalence is referred to as *observational equivalence*.

For additional material on automata, see, for example, [12, 18].

3.3 Rewriting Systems

In this paper we are concerned with very special rewriting systems. Information about general rewriting systems can be found in [4].

Let Σ be an alphabet (finite or infinite). A *rewriting system* over Σ consists of a set $\mathbf{T} \subseteq \Sigma^* \times \Sigma^*$ of *transformations* or *rules*. A *transformation* $(u, v) \in \mathbf{T}$ is written as $u \models v$. Then $\models^*$ is the reflexive and transitive closure of $\models$. Thus, $w \models^* w'$ (w derives w') if and only if $w = w_0 \models w_1 \models w_2 \models \cdots \models w_n = w'$ for some $n \geq 0$, and n is the length of this derivation of w' from w. In the special cases considered in this paper, the transformations have the pattern $u \models v$ or $ux \models vx$, where $u, v \in \Sigma^*$ are specific words and x is an arbitrary word in Σ^*.

A rewriting system is *confluent* if, for any $w, w_1, w_2 \in \Sigma^*$ with $w \models^* w_1$ and $w \models^* w_2$, there is $w' \in \Sigma^*$ such that $w_1 \models^* w'$ and $w_2 \models^* w'$. It is *Noetherian* if there is no word w from which a derivation of infinite length exists. A confluent Noetherian system has the following important property: For every word $w \in \Sigma^*$ there is a unique word $\tau(w)$, such that, for any $u \in \Sigma^*$ with $w \models^* u$, one has $u \models^* \tau(w)$ and there is no word $v \in \Sigma^*$ with $\tau(w) \models v$. For an effectively defined confluent Noetherian system, one can compute $\tau(w)$ for every word w.

4 Arbitrary Sets of Canonical Words

In this section we make no assumptions about the nature of the set of canonical words. We present a simple algorithm for finding generators for the state-equivalence relation of a given semiautomaton. Directly from the generators, we determine a rewriting system which transforms any word to its canonical representative.

Recall that we are dealing with connected semiautomata. Let $S = (\Sigma, Q, \delta, q_\epsilon)$ be a semiautomaton, and $\chi : Q \to \Sigma^*$, an arbitrary mapping assigning to state q a word $\chi(q)$ such that $\delta(q_\epsilon, \chi(q)) = q$. By definition χ is injective (one-to-one). *Unless stated otherwise, we assume that χ has been selected.* For $w \in \Sigma^*$, we call the word $\chi(q_w)$ the *canonical word* of state q_w, and the *canonical representative* of word w. Let the set of canonical words be $\mathbf{X}$.

4.1 Generators for State-Equivalence

We define a set $\mathbf{G}$ of pairs of state-equivalent words as follows. Consider the extension wa of each canonical word w by each letter a of Σ. If wa is canonical, there is no contribution to $\mathbf{G}$. Otherwise, add the pair $(wa, \chi(q_{wa}))$ to $\mathbf{G}$, to show that the canonical equivalent of wa is $\chi(q_{wa})$, the canonical word of the state reached by wa. We then augment $\mathbf{G}$ to obtain $\hat{\mathbf{G}}$, by declaring that the empty word is equivalent to the canonical word of the initial state. The set $\hat{\mathbf{G}}$ defines an equivalence relation $\equiv$ on Σ^* as follows.

Definition 1. *Relation $\equiv$ on Σ^* is the smallest right congruence containing the set $\hat{\mathbf{G}} = \mathbf{G} \cup \{(\epsilon, \chi(q_\epsilon))\}$, where $\mathbf{G}$ is the set of all ordered pairs $(wa, \chi(q_{wa}))$, with $w \in \mathbf{X}$, $a \in \Sigma$, and $wa \notin \mathbf{X}$.*

We refer to the pairs in $\mathbf{G}$ as *basic equivalences*. The number of basic equivalences is infinite in general; it is finite when Q and Σ are finite. In the sequel, we write the pairs in $\mathbf{G}$ as equivalences, that is, $wa \equiv \chi(q_{wa})$; moreover, we label the pairs by $\mathbf{E1}, \mathbf{E2}, \ldots$

For finite semiautomata, we can calculate the number of equations in $\mathbf{G}$ as follows [5].

Proposition 1. *Let S be a finite semiautomaton with n states and k input letters, and let $\mathbf{X}$ be a set of canonical words for S. Let n_0 be the number of words $w \in \mathbf{X}$ such that $w = ua$ with $a \in \Sigma$ and $u \in \mathbf{X}$. Then the number of equations in $\mathbf{G}$ is $nk - n_0$.*

Note that $0 \leq n_0 \leq n - 1$. If $\mathbf{X}$ is a prefix code, then $n_0 = 0$, because no extension wa of a canonical word w is canonical. At the other extreme, if $\mathbf{X}$ is prefix-closed then $n_0 = n - 1$, because, of the n canonical words, only the empty word is not an extension of a canonical word by a letter.

We define a set $\mathbf{T}$ of *basic transformations* as follows. If $w \equiv w'$ is a pair $\mathbf{Ei}$ in $\mathbf{G}$, then $wx \models w'x$ is the corresponding basic transformation $\mathbf{Ti}$ in $\mathbf{T}$. In these transformations, w and w' are fixed words and x is any word.

Example 1. Consider the semiautomaton of Fig. 1. The initial state is indicated by an incoming arrow, and each transition between two states is labelled by the input causing the transition.

Suppose $\chi(q_\epsilon) = \epsilon$, $\chi(q_0) = 01$, and $\chi(q_1) = 1$. Then we have the following basic equivalences and corresponding basic transformations for all $x \in \Sigma^*$:

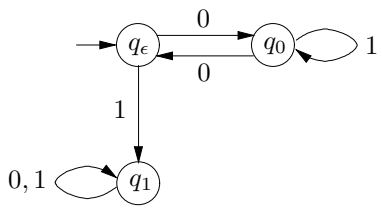

Fig. 1. Semiautomaton S_1

E1 $0 \equiv 01$, **E2** $10 \equiv 1$, **E3** $11 \equiv 1$, **E4** $010 \equiv \epsilon$, **E5** $011 \equiv 01$.
T1 $0x \models 01x$, **T2** $10x \models 1x$, **T3** $11x \models 1x$, **T4** $010x \models x$, **T5** $011x \models 01x$.

On the other hand, let $\chi(q_\epsilon) = 00$, $\chi(q_0) = 0$, and $\chi(q_1) = 1$. Then we have the following:

E1 $01 \equiv 0$, **E2** $10 \equiv 1$, **E3** $11 \equiv 1$, **E4** $000 \equiv 0$, **E5** $001 \equiv 1$.
T1 $01x \models 0x$, **T2** $10x \models 1x$, **T3** $11x \models 1x$, **T4** $000x \models 0x$, **T5** $001x \models 1x$.

Note that ϵ cannot derive $\chi(q_\epsilon) = 00$ in the latter **T**. □

Our goal in this subsection is to prove that $\equiv$ and $\equiv_\delta$ are equal. The inclusion $\equiv \subseteq \equiv_\delta$ is an immediate consequence of the definitions. We prove the converse containment with the aid of a rewriting system. First, we require some properties of **T** proved in [5].

Lemma 1.

1. *Rewriting according to* **T** *preserves* $\equiv$*, and hence also* $\equiv_\delta$*.*
2. *A word* w *derives* $\chi(q_w)$ *if and only if* w *has a canonical prefix.*
3. *If* w *has a canonical prefix and derives* w'*, then* w' *also has a canonical prefix.*
4. *No rule of* **T** *is applicable to a word which does not have a canonical prefix.*

Definition 2. *Given a set* $\mathbf{X}$ *of canonical words, we define the following subsets:*

- $\mathbf{W} = \Sigma^* \setminus \mathbf{X}\Sigma^*$ *is the set of* acanonical *words.*
- $\mathbf{X}_0 = \mathbf{X} \setminus \mathbf{X}\Sigma^+$ *is the set of* minimal canonical *words.*
- $\mathbf{Y} = \mathbf{X}_0\Sigma^+$ *is the set of* post-canonical *words.*

Set $\mathbf{W}$ consists of all the words that do not have a canonical prefix; clearly, $\mathbf{W}$ is prefix-closed. Set $\mathbf{X}_0$ is the set of canonical words w such that w has no canonical prefix other than w. This set is a prefix code. Set $\mathbf{Y}$ is the set of all words w such that w has at least one canonical prefix and is not in $\mathbf{X}_0$. Note that both $\mathbf{Y}$ and $\mathbf{X}_0 \cup \mathbf{Y}$ are prefix-continuous. The triple $(\mathbf{W}, \mathbf{X}_0, \mathbf{Y})$ is a partition of Σ^*. In general, all three sets may be infinite.

Theorem 1. $\equiv = \equiv_\delta$.

Proof. We know that $\equiv \subseteq \equiv_\delta$. To prove the converse, we show that $q_w = q_{w'}$ implies $w \equiv w'$, for all $w, w' \in \Sigma^*$. We do this by showing that each word w is equivalent to its canonical representative. From $q_w = q_{w'}$ it then follows that $w \equiv \chi(q_w) = \chi(q_{w'}) \equiv w'$.

We first claim that each acanonical word is equivalent to its canonical representative. Suppose ϵ is acanonical. Since the pair $(\epsilon, \chi(q_\epsilon))$ is in $\hat{\mathbf{G}}$, $\epsilon \equiv \chi(q_\epsilon)$. So the claim holds for the acanonical word of length 0. Now suppose that the claim holds for all acanonical words of length less than or equal to h, $h \geq 0$. Consider acanonical wa, where $|w| = h$, and $a \in \Sigma$. By the induction hypothesis, $w \equiv \chi(q_w)$. Since $\equiv$ is a right congruence, we have $wa \equiv \chi(q_w)a$. If $\chi(q_w)a$ is canonical, then $\chi(q_w)a = \chi(q_{wa})$, and $wa \equiv \chi(q_{wa})$. Otherwise, by construction of $\mathbf{G}$, the pair $(\chi(q_w)a, \chi(q_{wa}))$ is in $\mathbf{G}$, and the claim follows by transitivity of $\equiv$.

Next, consider a word w in $\mathbf{X}_0 \cup \mathbf{Y}$. Then $w \models^* \chi(q_w)$, and $w \equiv \chi(q_w)$ by Lemma 1 (2) and (1). This completes the proof. □

As a consequence of the theorem, the equivalence of two words is characterized by the equality of their canonical representatives.

Corollary 1. *For $w, w' \in \Sigma^*$, we have $\chi(q_w) = \chi(q_{w'})$ if and only if $w \equiv w'$.*

4.2 Transformations to Canonical Representatives

It is a disadvantage of the rewriting system $\mathbf{T}$ that an acanonical word cannot derive its canonical representative. To remedy this, we augment $\mathbf{T}$ to $\hat{\mathbf{T}}$, paralleling the construction of $\hat{\mathbf{G}}$ from $\mathbf{G}$.

Definition 3. $\hat{\mathbf{T}} = \mathbf{T} \cup \{w \models \chi(q_\epsilon)w \mid w \in \mathbf{W}\}$.

We call the added rules *acanonical*. Note that acanonical rule $w \models \chi(q_\epsilon)w$ can be applied only to the acanonical word w. After this rule is applied, the result is a post-canonical word. No acanonical rule is applicable after the first step.

With the added rules, the rewriting system $\hat{\mathbf{T}}$ always permits the derivation of the canonical representative of any word, as proved in [5]:

Lemma 2. *The rewriting systems $\mathbf{T}$ and $\hat{\mathbf{T}}$ are confluent, and every $w \in \Sigma^*$ derives its canonical representative $\chi(q_w)$ in $\hat{\mathbf{T}}$.*

The main result concerning the rewriting system can now be proved.

Theorem 2. *The equivalence of two words in Σ^* is provable by transformations in the rewriting system $\hat{\mathbf{T}}$.*

Proof. By Corollary 1, $w \equiv w'$ if an only if the canonical representatives of w and w' are identical. By Lemma 2, every word derives its canonical representative in $\hat{\mathbf{T}}$. □

4.3 Semiautomata Versus Canonical Words and Equivalences

The original goal of trace-assertions was to avoid all implementation aspects in the abstract specification of a module. Specification by automata was considered to be too close to implementation. It appears that when traces are used there is no need to identify the states of the automaton. Our results show, however, that complete state information is implicit in a trace-assertion specification. More precisely, a semiautomaton can be reconstructed from its canonical words and equivalences.

Let $S = (\Sigma, Q, \delta, q_\epsilon)$ be a semiautomaton, let $\mathbf{X}$ be a set of canonical words, and let $\hat{\mathbf{G}}$ be the set of equivalences derived from S. Let $S_{\mathbf{X}} = (\Sigma, \mathbf{X}, \delta_{\mathbf{X}}, \chi(q_\epsilon))$ be the semiautomaton in which, for all $w \in \mathbf{X}, a \in \Sigma$, $\delta_{\mathbf{X}}(w, a) = wa$ if $wa \in \mathbf{X}$, and $\delta_{\mathbf{X}}(w, a) = \chi(q_{wa})$, if $(wa, \chi(q_{wa})) \in \mathbf{G}$. The semiautomata S and $S_{\mathbf{X}}$ are isomorphic [5]. The isomorphism maps state $q \in Q$ to canonical word $\chi(q) \in \mathbf{X}$.

In summary, the information contained in a specification by canonical words and equivalences is precisely the same as that in the semiautomaton in which the canonical words have been selected. Consequently, one can view the semiautomaton as *the specification*, and the various sets of canonical words and the corresponding equivalences as *implementations*, in the following sense. In a specification by a semiautomaton, the state labels can be picked arbitrarily, and changed at will, without affecting the semiautomaton, whereas, in a specification by canonical words and equivalences, one makes a commitment to a particular representation of states.

5 Prefix-Continuous Sets of Canonical Words

All the results of the previous section hold for arbitrary canonical sets. Equivalence of two words w and w' is provable in $\hat{\mathbf{T}}$ by Theorem 2. However, we still have the problem that the rewriting system may permit infinite derivations.

Example 2. Return to the semiautomaton of Fig. 1, with $\chi(q_\epsilon) = \epsilon$, $\chi(q_0) = 01$, and $\chi(q_1) = 1$, and the corresponding rules:

T1 $0x \models 01x$, **T2** $10x \models 1x$, **T3** $11x \models 1x$, **T4** $010x \models x$, **T5** $011x \models 01x$.

We have the following derivation starting at 0 and leading to its canonical representative:

$$0 \overset{\mathbf{T1}}{\models} 01.$$

Note, however, that rule **T1** can be applied repeatedly, leading to the derivation

$$0 \overset{\mathbf{T1}}{\models} 01 \overset{\mathbf{T1}}{\models} 011 \overset{\mathbf{T1}}{\models} 0111 \overset{\mathbf{T1}}{\models} \ldots,$$

which never terminates. There is yet another derivation

$$0 \overset{\mathbf{T1}}{\models} 01 \overset{\mathbf{T1}}{\models} 011 \overset{\mathbf{T5}}{\models} 01 \overset{\mathbf{T1}}{\models} 011 \overset{\mathbf{T5}}{\models} 01\ldots,$$

which is also infinite. □

5.1 Deterministic Derivations

We now show that, if $\mathbf{X}$ is prefix-continuous, infinite derivations are impossible. The proofs of the results below can be found in [5].

Lemma 3. *If* $\mathbf{X}$ *is prefix-continuous, the set* $\mathbf{L}$ *of all left-hand sides of the generating equivalences in* $\mathbf{G}$ *is a prefix code.*

Lemma 4. *If* $\mathbf{L}$ *is a prefix code, at most one rule of* $\hat{\mathbf{T}}$ *applies to any word, and, if* $w \in \mathbf{X}$*, no rule of* $\hat{\mathbf{T}}$ *applies to* w*.*

Lemma 4 implies that rewriting in $\hat{\mathbf{T}}$ is deterministic. By Lemma 2, a word derives its canonical representative, and no rule applies to the canonical representative. Thus, the canonical representative is the $\tau(w)$ of Subsection 3.3.

Theorem 3. *The rewriting system* $\hat{\mathbf{T}}$ *is Noetherian if and only if the set* $\mathbf{X}$ *of canonical words is prefix-continuous.*

Proof. Suppose $\mathbf{X}$ is prefix-continuous. By Lemma 3, $\mathbf{L}$ is a prefix code. By Lemma 4, at most one rule applies to any word. Hence the rewriting process is deterministic. By Lemma 2, each word derives its canonical representative, from which no further derivation is possible, by Lemma 4. Therefore $\hat{\mathbf{T}}$ is Noetherian.

Conversely, suppose that $\mathbf{X}$ is not prefix-continuous. Then there exists $w = uav \in \mathbf{X}$ such that $u \in \mathbf{X}$, but $ua \notin \mathbf{X}$. Therefore $(ua, \chi(q_{ua})) \in \mathbf{G}$, and $w = uav \models \chi(q_{ua})v$. By Lemma 1 (1), w and $\chi(q_{ua})v$ lead to the same state. By Lemma 2, $\chi(q_{ua})v \models^* \chi(q_w) = w$. Thus $w \models \chi(q_{ua})v \models^* w$, and the rewriting system is not Noetherian. □

This result implies that trace-assertion specifications must use prefix-continuous sets of canonical words; otherwise infinite derivations are inevitable.

Theorem 4. *If* $\mathbf{X}$ *is prefix-continuous, then* $\hat{\mathbf{G}}$ *is irredundant, that is:*

- *If* $\epsilon \notin \mathbf{X}$*, then* $\mathbf{G}$ *does not generate* $\equiv_\delta$*.*
- *For any pair* $p = (ua, \chi(q_{ua})) \in \mathbf{G}$*, the set* $\hat{\mathbf{G}} \setminus p$ *does not generate* $\equiv_\delta$*.*

This theorem, proved in [5], shows that one cannot reduce the number of equivalences in a trace-assertion specification in which derivations are finite.

5.2 Prefix-Continuous Sets and Spanning Forests

Prefix-continuous canonical sets can be found with the aid of certain graph-theoretic constructs. Recall that a directed graph is a pair $G = (V, E)$, where V is the set of vertices of G and $E \subseteq V \times V$ is the set of (directed) edges of G. A *spanning forest* of a directed graph $G = (V, E)$ is a set of pairwise disjoint trees, such that V is the union of all the vertices in the trees. A spanning tree is a spanning forest consisting of a single tree.

To find a prefix-continuous canonical set for a semiautomaton S, we can use a spanning forest. Given such a forest of disjoint trees, for the root r of a

tree, choose an arbitrary word w_r leading to state r from the initial state of S. Proceeding by induction, if state q has been assigned word w_q and state q' is a child of q reached from q by applying input a, then state q' is assigned word $w_q a$. In this way we associate a word with each state of S. The set of these words is then the set of canonical words for S, and it is prefix-continuous.

Example 3. Consider the semiautomaton of Fig. 1, and the forest of three one-vertex trees $\{q_\epsilon\}$, $\{q_0\}$ and $\{q_1\}$. We can choose 00, 01, and 1 for the roots $\{q_\epsilon\}$, $\{q_0\}$ and $\{q_1\}$, respectively, resulting in the set $\{00, 01, 1\}$ of canonical words. This set is a prefix code. The acanonical words are ϵ and 0, and the set of post-canonical words is $\{00, 01, 1\}\Sigma^+$.

On the other hand, we can choose the trees with vertices $\{q_1\}$ and $\{q_\epsilon, q_0\}$. If we pick q_1 and q_0 as roots, and assign 1 to q_1, and 0 to q_0, then q_ϵ is assigned 00, and $\mathbf{X} = \{1, 0, 00\}$.

We can also choose a single tree with vertices $\{q_\epsilon, q_0, q_1\}$ and root at q_0. If we assign 0 to the root, then q_ϵ and q_1 are assigned 00 and 001, respectively. □

Conversely, given a prefix-continuous canonical set $\mathbf{X}$, we can construct a spanning forest for S. The states reached from the initial state by the minimal canonical words are the roots of the forest. Continuing by induction, if word $u \in \mathbf{X}$ corresponds to state q, and if $a \in \Sigma$ and $ua \in \mathbf{X}$, then q_{ua} is a child of q under input a. Thus to each word in $\mathbf{X}$ we associate a vertex in the forest; this is possible because $\mathbf{X}$ is prefix-continuous.

5.3 Special Canonical Sets

The family of prefix-continuous canonical sets contains two extreme special cases: prefix-closed sets and prefix codes. Prefix-closed sets are widely applicable, as our later examples show.

To find a prefix-closed set of canonical words we can use a spanning tree of the state graph of the semiautomaton S, with q_ϵ as root, and $\chi(q_\epsilon) = \epsilon$.

Example 4. Consider the semiautomaton S_2 of Fig. 2. We show three spanning trees for S_2. The basic equivalences corresponding to the three spanning trees are, by rows,

E1 $01 \equiv 1$,	**E2** $10 \equiv 00$,	**E3** $11 \equiv 1$,	**E4** $000 \equiv 1$,	**E5** $001 \equiv 0$.
E1 $1 \equiv 01$,	**E2** $00 \equiv 010$,	**E3** $011 \equiv 01$,	**E4** $0100 \equiv 01$,	**E5** $0101 \equiv 0$.
E1 $00 \equiv 10$,	**E2** $01 \equiv 1$,	**E3** $11 \equiv 1$,	**E4** $100 \equiv 1$,	**E5** $101 \equiv 0$. □

Our next example illustrates the usefulness of prefix codes as canonical sets. A similar example was suggested to us by David L. Parnas; the specific semiautomaton we use here is its simplified and modified version.

Example 5. Consider the 2-bit shift register of Fig. 3, started in state $(y_1, y_2) = (0, 0)$, with binary input x. The register contents are shifted to the left, with the value of x shifted to y_2 and the value of y_2 shifted to y_1. Assume that the

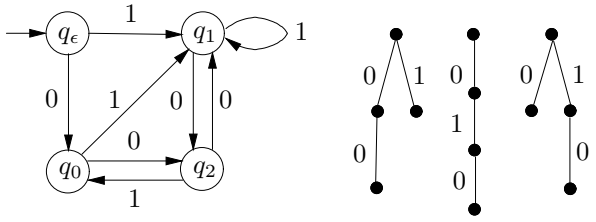

Fig. 2. Semiautomaton S_2 and spanning trees

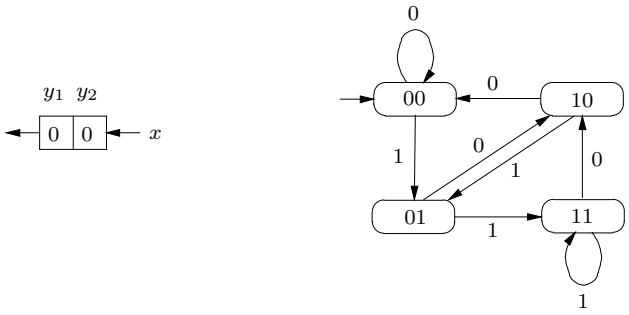

Fig. 3. 2-bit shift register

shifts occur at integral values of time: $1, 2, \ldots t, \ldots$ Thus, at time $t+1$, we have $y_2(t+1) = x(t)$ and $y_1(t+1) = y_2(t)$. The semiautomaton of the shift register is shown in the figure, with parentheses and commas omitted from the state tuples for simplicity.

A possible representation for the states of the register is shown in the figure, where each state represents the register contents. The set of basic equivalences is: $\{000 \equiv 00, 001 \equiv 01, 010 \equiv 10, 011 \equiv 11, 110 \equiv 10, 111 \equiv 11, 100 \equiv 00, 101 \equiv 01\}$. The set $\{00, 01, 10, 11\}$ of canonical words has the advantage of using the natural state representation, and has much symmetry. For example, all eight rewriting rules can be summarized in one statement:

$$abc \equiv bc, \quad \forall a, b, c \in \Sigma.$$

This symmetry is lost if a prefix-closed set is used. □

6 Counter

We now present our first example of an infinite semiautomaton, and the concept of legality. This concept was introduced in [1] to distinguish the normal operation of a module from its behavior when abnormal conditions occur. In later works on trace-assertion specifications (for example, [20]) this concept was abandoned.

We prefer to retain it, however, as an optional feature of a specification. Legality provides a convenient example of the use of final (accepting) and non-final (rejecting) states of an automaton to separate two types of behavior. In general, one may use a Moore output with more than two values to partition the states into several classes of behaviors.

6.1 Counter with Empty Initial State

The initial count is 0. Only two operations are possible: INCREMENT, denoted by 1, and DECREMENT, denoted by 0. If the count is 0, DECREMENT is illegal and leads to a special illegal state.[1] In any legal state it is possible to INCREMENT the count by 1. If the count is $(n+1)$, where $n \geq 0$, DECREMENT subtracts 1 from the count.

Definition 4. *The* counter automaton *is* $A = (\Sigma, Q, \delta, q_\epsilon, F)$*, where* $\Sigma = \{0, 1\}$*,* $Q = P \cup \{\infty\}$*,* $q_\epsilon = 0$*,* $F = P$*, and* δ *is defined* [2] *as*

$$\begin{array}{ll} \mathbf{C1'} & \delta(n, 1) = n + 1, \quad \forall n \in P, \\ \mathbf{C2'} & \delta(0, 0) = \infty, \\ \mathbf{N1'} & \delta(\infty, a) = \infty, \quad \forall a \in \Sigma, \\ \mathbf{N2'} & \delta(n + 1, 0) = n, \quad \forall n \in P. \end{array}$$

The state graph of A is shown in Fig. 4 (a), where vertices drawn with thick lines indicate final states. It should be clear that the automaton corresponds to our informal specification.

It seems reasonable that a specification of a module should use a *reduced* automaton. Otherwise, unnecessary states and transitions are introduced. Note, however, that our theory applies equally well to non-reduced automata. In a reduced automaton every two distinct states are in different classes of the Nerode equivalence, that is, they are observationally inequivalent.

Proposition 2. *The counter automaton is reduced.*

Proof. State ∞ is distinguishable from every other state, because it is the only rejecting state. To distinguish state n from state $m > n$, use the word 0^m. Then $\delta(n, 0^m) = \infty \notin F$ and $\delta(m, 0^m) = 0 \in F$. □

In the case of the counter, the representation of the legal states by non-negative integers is very natural. It is, however, also quite natural to represent states by canonical traces instead. Automata with states represented by traces are said to be in *standard form*. Some examples given below, show that the standard form is not always the most natural one, and that its construction may require considerable effort.

[1] In general, one could have several illegal states representing various error conditions, as shown in Section 9.

[2] The reason for the particular labelling of items will become apparent later.

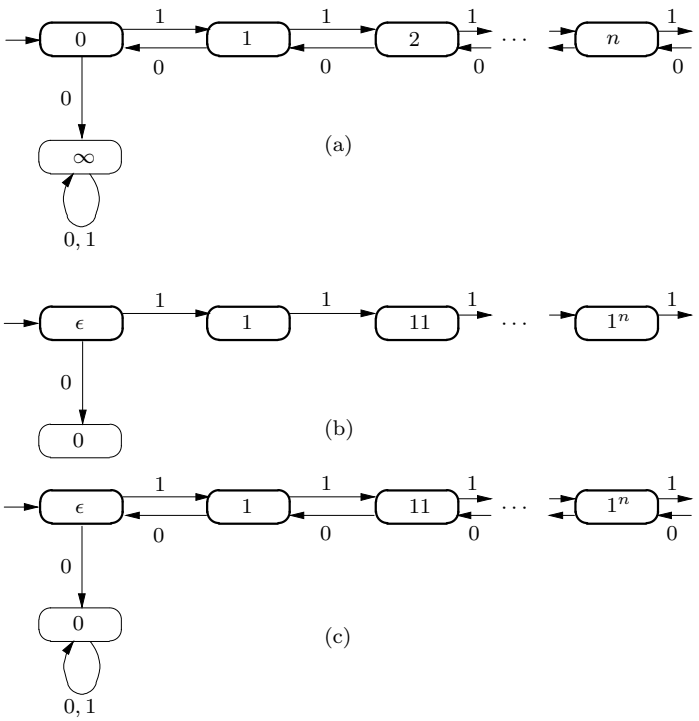

Fig. 4. Counter automaton and canonical traces

The first step in constructing a trace-assertion specification is to select canonical traces. For the counter, a natural choice is 1^n to represent the count of n, and 0 for the illegal DECREMENT. The set $\{1\}^* \cup \{0\}$ is prefix closed. This step is illustrated in Fig. 4 (b). In the semiautomaton of Fig. 4 (a), this choice of canonical traces corresponds to a spanning tree.

The second step consists of finding the set **G** of basic equivalences. These equivalences provide the missing transitions in Fig. 4 (b), resulting in Fig. 4 (c).

The basic equivalences and the corresponding basic transformations are

$$\begin{array}{llllllll} \mathbf{E1'} & 00 \equiv 0 & \mathbf{E2'} & 01 \equiv 0 & \mathbf{E3'} & 10 \equiv \epsilon & \mathbf{E4'} & 110 \equiv 1 \quad \ldots \\ \mathbf{T1'} & 00x \models 0x & \mathbf{T2'} & 01x \models 0x & \mathbf{T3'} & 10x \models x & \mathbf{T4'} & 110x \models 1x \quad \ldots \end{array}$$

The set of equivalences is, of course, infinite. However, we can represent this infinite set by two patterns:

$$\begin{array}{ll} \mathbf{E1} & 0a \equiv 0, \qquad \forall a \in \Sigma, \\ \mathbf{E2} & 1^{n+1}0 \equiv 1^n, \quad \forall n \in P. \end{array}$$

In fact, if we relabel the states with their canonical representatives, the definition of δ becomes

$$\begin{array}{ll} \textbf{C1} & \delta(1^n,1)=1^{n+1}, \quad \forall n\in P,\\ \textbf{C2} & \delta(\epsilon,0)=0,\\ \textbf{N1} & \delta(0,a)=0, \qquad \forall a\in\Sigma,\\ \textbf{N2} & \delta(1^{n+1},0)=1^n, \quad \forall n\in P. \end{array}$$

Now there is a 1-1 correspondence between the **Ni** and the **Ei**. Equations **Ni** correspond to non-canonical extensions of canonical traces by letters. Equations **Ci** correspond to canonical extensions of canonical traces by letters; hence they do not contribute to the equivalences.

We are now in a position to state the complete set of trace assertions for the counter. Following [1], we add *syntax* and *legality* sections. The syntax assertions are type declarations. Each operation, that is, each element of Σ, results in a state transition; thus it maps type $\langle$counter$\rangle$ into type $\langle$counter$\rangle$.

For $w\in\Sigma^*$, the assertion "$\lambda(w)=$ true" means that w is a legal trace. All the canonical traces in $\{1\}^*$ are declared legal by **L1** below, and they correspond to the final states of the automaton. The remaining legal traces are obtained by the assertion:

$$\textbf{L0} \quad u\equiv v\Rightarrow\lambda(u)=\lambda(v), \quad \forall u,v\in\Sigma^*,$$

which is assumed to hold in every trace-assertion specification. Finally, no trace is legal, unless its being so is a consequence of **L0** and **L1**. Thus the set of legal traces is the smallest set containing the legal canonical traces, and closed under **L0**.

Combining all the parts, we obtain the specification:

Syntax:
$0,1:\langle\text{counter}\rangle\to\langle\text{counter}\rangle$.
Canonical traces:
$\{1\}^*\cup\{0\}$.
Equivalence:
$$\begin{array}{ll} \textbf{E1} & 0a\equiv 0, \qquad \forall a\in\Sigma,\\ \textbf{E2} & 1^{n+1}0\equiv 1^n, \quad \forall n\in P. \end{array}$$
Legality:
$$\textbf{L1} \quad \lambda(1^n)=\text{true}, \quad \forall n\in P.$$
Transformations:
$$\begin{array}{ll} \textbf{T1} & 0ax\models 0x, \qquad \forall a\in\Sigma, x\in\Sigma^*\\ \textbf{T2} & 1^{n+1}0x\models 1^nx, \quad \forall n\in P, x\in\Sigma^*. \end{array}$$

There is no "values" part, since there are no output-producing operations in our counter. Outputs will be handled in the next section. Note also that transformation **T1** can be simplified to $0x\models 0$, for all $x\in\Sigma^+$.

6.2 Counter with Nonempty Initial State

Suppose that we want to change the initial state from the empty state to the state that contains two 1s. In the specification by automaton, this operation is

entirely trivial. For example, in the automaton of Fig. 4 (a), instead of $A = (\Sigma, Q, \delta, 0, F)$, we now use $A = (\Sigma, Q, \delta, 2, F)$, and, for Fig. 4 (c), instead of $A = (\Sigma, \mathbf{X}, \delta_{\mathbf{X}}, \epsilon, 1^*)$, we have $A = (\Sigma, \mathbf{X}, \delta_{\mathbf{X}}, 11, 1^*)$. In the trace-assertion specification, however, we need to find a new spanning forest, and recalculate the equivalences.

In Fig. 5 (a), we show the solution using the spanning tree corresponding to the canonical set $\{\epsilon, 0, 00, 000, 1, 11, 111, \ldots\}$. This solution has the disadvantage that the state label no longer corresponds to the present count. Also, we must calculate a new set of equivalences, in this case:

E1 $01 \equiv \epsilon$,
E2 $001 \equiv 0$,
E3 $000a \equiv 000, \quad \forall a \in \Sigma$,
E4 $1^{n+1}0 \equiv 1^n, \quad \forall n \in P$.

A second solution is shown in Fig. 5 (b), where we use two trees corresponding to the two sets of canonical traces: $\{00, 000, 001\}$ and $\{11, 111, \ldots\}$. The advantage of this solution is that, except for three states, the state label corresponds to the contents of the counter. Now the equivalences are

E1 $110 \equiv 001$,
E2 $0011 \equiv 11$,
E3 $0010 \equiv 00$,
E4 $000a \equiv 000, \quad \forall a \in \Sigma$,
E5 $1^{n+1}0 \equiv 1^n, \quad \forall n \geq 2$.

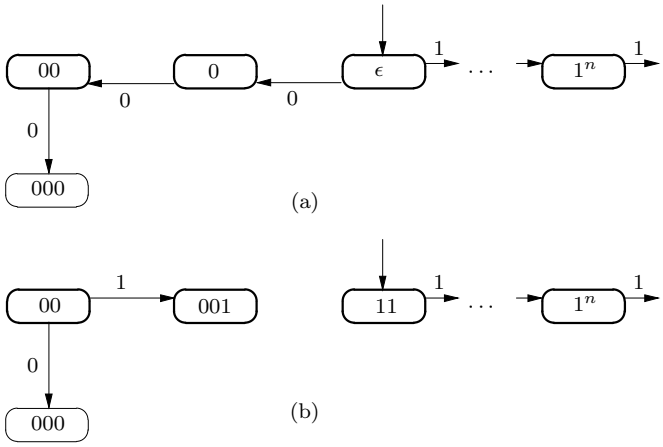

Fig. 5. Counter automaton with changed initial state

To complete the specification, we must add the rule $\epsilon \equiv 11$ to take care of the new initial state. The acanonical traces are $\epsilon \cup 0 \cup 1 \cup (01 \cup 10)\Sigma^*$. To

find the canonical representative of any nonempty acanonical trace w, we use the right-congruence property: $\epsilon \equiv 11$ implies $w \equiv 11w$, and then apply the transformation rules from **T** to $11w$, which is post-canonical. This approach would require us to test the given trace for membership in the set of acanonical traces. Alternatively, one can put $\chi(q_\epsilon)$ in front of *any* trace, and then derive the canonical representative as above, thus avoiding the membership test, at the cost of at most one extra step in the derivation.

7 Stack

In this section we introduce a more general module, one that has an infinite alphabet and output operations, called "value functions" in [1].

The stack is initially empty. We can push any integer z onto the stack using operation PUSH(z), denoted by z. The POP operation p, legal only if the stack is nonempty, removes the top integer from the stack. The TOP operation t, legal only if the stack is nonempty, returns the value of the top integer. If the stack is empty, p and t lead to the illegal state. The DEPTH operation d returns the number of integers stored on the stack, when it is in any legal state.

We use the word $z_1 \dots z_n$ as the canonical word[3] for the state of the stack when it contains $(z_1, \dots, z_n)$. Let p be the canonical trace for the illegal state. Clearly, $Z^* \cup \{p\}$ is prefix closed.

Definition 5. *The* stack automaton *is a generalized Mealy automaton* $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$, *where* $\Sigma = \{d, p, t\} \cup Z$, $Q = Z^* \cup \{p\}$, $q_\epsilon = \epsilon$, $F = Z^*$, $\Omega = Z$, *and* δ *and* ν *are defined below. Note that* $\nu = \nu(q, a)$ *is defined only if* $q \in Z^*$ *and* $a = d$, *or* $q \in Z^+$ *and* $a = t$.

$$
\begin{array}{ll}
\textbf{C1} & \delta(q, z) = qz, \quad \forall q \in Z^*, z \in Z, \\
\textbf{C2} & \delta(\epsilon, p) = p, \\
\textbf{N1} & \delta(\epsilon, t) = p, \\
\textbf{N2} & \delta(q, d) = q, \quad \forall q \in Z^*, \\
\textbf{N3} & \delta(p, a) = p, \quad \forall a \in \Sigma, \\
\textbf{N4} & \delta(qz, t) = qz, \quad \forall q \in Z^*, z \in Z, \\
\textbf{N5} & \delta(qz, p) = q, \quad \forall q \in Z^*, z \in Z, \\
\textbf{O1} & \nu(q, d) = |q|, \quad \forall q \in Z^*, \\
\textbf{O2} & \nu(qz, t) = z, \quad \forall q \in Z^*, z \in Z.
\end{array}
$$

The stack automaton is illustrated in Fig. 6. For state q and input a, the transition from q under a is labelled by a, if there is no output. If there is an output b, the transition is labelled by (a, b). Of course, we can only illustrate several typical transitions, since both Q and Σ are infinite. There is one transition from each state for each of d, p, and t, and for each integer z. Note that d never changes the state, and t changes it only when illegally applied. For $q \in Z^*$, $\nu(q, d) = |q|$ is the number of integers on the stack, and $\nu(qz, t) = z$ is the top integer.

[3] In the figure, we use the notation $(z_1, \dots, z_n)$ to avoid confusion.

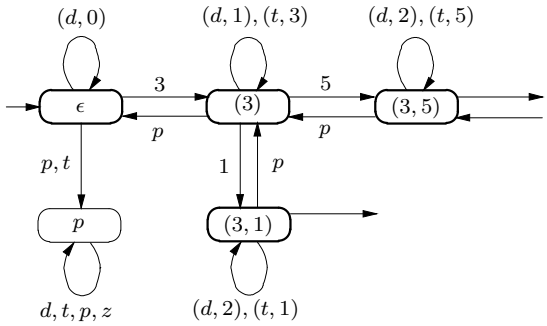

Fig. 6. Stack automaton

Proposition 3. *The stack automaton is reduced.*

Proof. State p is a rejecting state and all the states in Z^* are accepting. Suppose now that q and q' are accepting states, and $|q| = i$, $|q'| = j$, and $i < j$. Then q and q' are distinguishable by the trace p^j. Assume now that q and $q' \neq q$ are of equal length, and their longest common suffix is $q_{i+1} \dots q_n$; then $q_i \neq q'_i$. Then q and q' are distinguishable by $p^{n-i}t$. □

The basic equivalences are shown below as part of the complete trace-assertion specification. Equivalence $\equiv$ is the right congruence generated by the rules **E1**–**E5**. These rules are obtained as follows. The empty trace is canonical. Hence we examine all the traces of the form $\epsilon a = a$, with $a \in \Sigma$. If $a = z$, the extension is canonical; hence, there is no contribution to the equivalences from **C1**. If $a = p$, again the extension is canonical, and there is no contribution from **C2**. If $a = t$, we have the equivalence **E1** $t \equiv p$. If $a = d$, we obtain $d \equiv \epsilon$. However, this case can be handled with all the other cases of the form $wd \equiv w$, since the transition function has the value $\delta(q, d) = q$, for all $q \in Z^*$. Thus we obtain **E2**. For the illegal state, we obtain **E3** from **N3**. For all the canonical traces of the form qz, we again examine all the extensions by letters. The extension by another integer is already covered by **C1**. The extension by d is covered by **E2**. For t, we have **E4**, and for p, **E5**. Again, there is an obvious 1-1 correspondence between the **Ni** and the **Ei**.

Since the set of accepting states of M is $F = Z^*$, all the canonical traces in Z^* are declared legal by **L1** below. The remaining legal traces are obtained by

L0 $\quad u \equiv v \Rightarrow \lambda(u) = \lambda(v), \quad \forall u, v \in \Sigma^*.$

Until now, we have ignored the output values produced by operations t and d. With the aid of **O1** and **O2**, we specify the values for canonical legal traces, and then make the values applicable to all traces by the assertion

V0 : $\quad w \equiv w' \Rightarrow \nu(wa) = \nu(w'a), \quad \forall w, w' \in \Sigma^*, a \in \Sigma.$

We now state the complete set of trace assertions for the stack. Each element of $Z \cup \{p\}$ results in a state transition, that is, maps type ⟨stack⟩ into type

$\langle$stack$\rangle$. Moreover, inputs d and t also produce an output; those inputs map type $\langle$stack$\rangle$ into type $\langle$stack$\rangle \times \langle$integer$\rangle$. Note that a stack in an illegal state is considered to be of type $\langle$stack$\rangle$. Note also that d does not change the state of the stack; however, it is considered to be a state transition.

Since the syntax assertions are straightforward, we leave them to the reader in subsequent examples.

Syntax:
$p, z : \langle\text{stack}\rangle \to \langle\text{stack}\rangle, \qquad \forall z \in Z,$
$d, t : \langle\text{stack}\rangle \to \langle\text{stack}\rangle \times \langle\text{integer}\rangle.$

Equivalence:
E1 $t \equiv p,$
E2 $wd \equiv w,$
E3 $pa \equiv p, \quad \forall a \in \Sigma,$
E4 $wzt \equiv wz, \quad \forall w \in Z^*, z \in Z,$
E5 $wzp \equiv w, \quad \forall w \in Z^*, z \in Z.$

Legality:
L1 $\lambda(w) = \text{true}, \quad \forall w \in Z^*.$

Values:
V1 $\nu(wd) = |w|, \quad \forall w \in Z^*,$
V2 $\nu(wzt) = z, \quad \forall z \in Z, w \in Z^*.$

Transformations:
T1 $tx \models px,$
T2 $wdx \models wx,$
T3 $pax \models px, \quad \forall a \in \Sigma,$
T4 $wztx \models wzx, \quad \forall w \in Z^*, z \in Z,$
T5 $wzpx \models wx, \quad \forall w \in Z^*, z \in Z.$

•••

In the rest of our examples in the paper and its appendices we give only the annotated automaton definitions. We include these examples to illustrate the construction of specifications of modules by automata. If these automata are in standard form, we obtain trace-assertion specifications algorithmically. However, in four of our examples, namely set, maximal element module, linked list, and traversing stack, there are representations of states which are more natural than canonical traces. Thus, although trace-assertion specifications are always possible, considerable work may be required to obtain them.

In specifying modules by automata, we find it useful to draw partial state graphs. To simplify the figures, we omit the outputs and show only the transitions of the underlying semiautomata. These drawings help in deriving the formal definitions and in checking whether all cases have been considered.

8 Set

This example is derived from the "intset" example of [6], discussed also in [19]. We start with an empty set S. We can add any integer z to S using INSERT(z),

denoted by z; it does not change S if $z \in S$. DELETE(z), denoted by $\bar{z}$, removes z from S, and does nothing if $z \notin S$. MEMBER(z), denoted by $\dot{z}$, returns false if $z \notin S$, and true if $z \in S$.

Let $\bar{Z} = \{\bar{z} \mid z \in Z\}$, and $\dot{Z} = \{\dot{z} \mid z \in Z\}$. The obvious definition of a set automaton uses all finite sets of integers as states. The set semiautomaton is illustrated in Fig. 7.

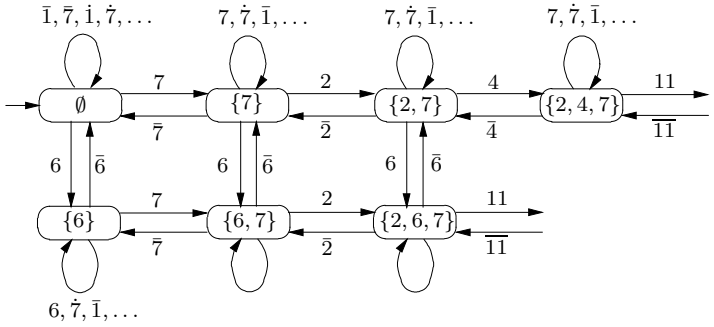

Fig. 7. Set semiautomaton

Definition 6. *The* set automaton *is* $M' = (\Sigma, Q', \delta', q'_\epsilon, F', \Omega, \nu')$*, where* $\Sigma = Z \cup \bar{Z} \cup \dot{Z}$*,* Q' *is the set of all finite subsets of* Z*,* $q'_\epsilon = \emptyset$*,* $F' = Q'$*,* $\Omega = \{\text{true}, \text{false}\}$*, and*

M1	$\delta'(q', z) = q' \cup \{z\}$,	$\forall q' \in Q', z \in Z$,
M2	$\delta'(q', \bar{z}) = q' \setminus \{z\}$,	$\forall q' \in Q', z \in Z$,
M3	$\delta'(q', \dot{z}) = q'$,	$\forall q' \in Q', z \in Z$,
O	$\nu'(q', \dot{z}) = z \in q'$,	$\forall q' \in Q', z \in Z$.

This definition is not in standard form, since the representative of a state is not a word in Σ^*. Furthermore, rule **M1** represents both the case where the extension leads to a new canonical state, and the case where the state does not change. To obtain a standard form we need to choose a new state representation.

Define the function $setsort : Q' \to Z^*$ as follows: $setsort(\emptyset) = \epsilon$, and if $q' = \{z_1, \ldots, z_n\} \in Q'$, $setsort(q')$ is the word that consists of $z_1, \ldots, z_n$ arranged in decreasing order. Note that the image $setsort(Q')$ is the set of all sorted words without repeated letters. Define function $set : Z^* \to Q'$ as follows. If $w = z_1 \ldots z_n \in Z^*$, then $set(w)$ is the set of $z \in Z$ appearing in w.

For $w \in Z^*$ and $z \in Z$, we write $z \in w$ if letter z appears in word w. We now represent states by words in $Q = setsort(Q')$. This set is prefix closed. For the canonical word of state $q' \in Q'$, we now choose $setsort(q')$. All words are legal. We denote by $\omega(q)$ the last letter of q if q is nonempty, and ∞ if $q = \epsilon$.

Definition 7. *The* standard set automaton *is* $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$*, where* $\Sigma = Z \cup \bar{Z} \cup \dot{Z}$*,* $Q = setsort(Q')$*,* $q_\epsilon = \epsilon$*,* $F = Q$*,* $\Omega = \{\text{true}, \text{false}\}$*, and*

C1	$\delta(q, z) = qz,$	$\forall q \in Q, z \in Z, z \notin q, z < \omega(q),$
N1	$\delta(q, z) = setsort(set(q) \cup \{z\}),$	$\forall q \in Q, z \in Z, z \notin q, z \geq \omega(q),$
N2	$\delta(q, z) = q,$	$\forall q \in Q, z \in Z, z \in q,$
N3	$\delta(q, \bar{z}) = setsort(set(q) \setminus \{z\}),$	$\forall q \in Q, z \in Z,$
N4	$\delta(q, \dot{z}) = q,$	$\forall q \in Q, z \in Z,$
O1	$\nu(q, \dot{z}) = \text{false},$	$\forall q \in Q, z \in Z, z \notin q,$
O2	$\nu(q, \dot{z}) = \text{true},$	$\forall q \in Q, z \in Z, z \in q.$

One verifies that the two automata are isomorphic and reduced.

This example illustrates that, in some cases, the representation of states by canonical words, although always possible, can be quite awkward.

9 Bounded Stacks

In practice, stacks are finite in two senses. First, the size of the stack is limited by some maximum capacity n. Second, the size of the integers is limited to some maximum value b.

For $b, n \in P$, let $B = \{z \mid 0 \leq z \leq b\}$, and let $B_n = \bigcup_{i=0}^{n} B^i$. It is illegal to push an integer if either that integer is not in B, or the stack is full, that is, has depth n. The stack automaton of Section 7 needs to be modified. For canonical representatives of legal states we choose $q \in B_n$, and for the illegal state we pick p.

The bounded stack semiautomaton is illustrated in Fig. 8, with $n = 2$, and $B = \{0, 1\}$, where we omit all inputs in $Z \setminus B$, which lead to p.

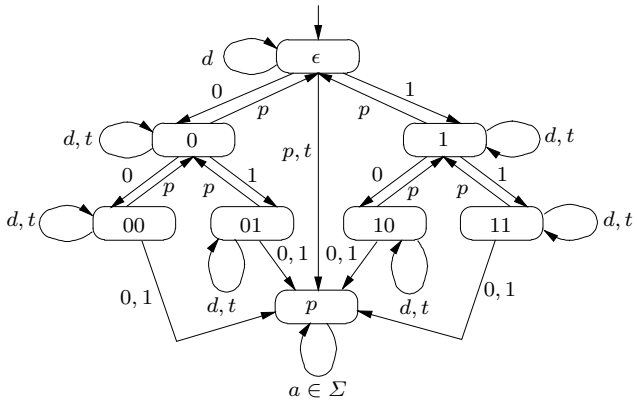

Fig. 8. Bounded stack semiautomaton

Definition 8. *The* bounded stack automaton *(with legal input set B and capacity n) is a generalized Mealy automaton $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$, where $\Sigma = \{d, p, t\} \cup Z$, $Q = B_n \cup \{p\}$, $q_\epsilon = \epsilon$, $F = B_n$, $\Omega = B \cup \{0, \ldots, n\}$, and*

C1	$\delta(q, z) = qz,$	$\forall q \in B_{n-1}, z \in B,$
C2	$\delta(\epsilon, p) = p,$	
N1	$\delta(q, z) = p,$	$\forall q \in B^n, z \in Z,$
N2	$\delta(q, z) = p,$	$\forall q \in B_{n-1}, z \in Z \setminus B,$
N3	$\delta(\epsilon, t) = p,$	
N4	$\delta(q, d) = q,$	$\forall q \in B_n,$
N5	$\delta(p, a) = p,$	$\forall a \in \Sigma,$
N6	$\delta(qz, t) = qz,$	$\forall q \in B_{n-1}, z \in B,$
N7	$\delta(qz, p) = q,$	$\forall q \in B_{n-1}, z \in B,$
O1	$\nu(q, d) = \lvert q \rvert,$	$\forall q \in B_n,$
O2	$\nu(qz, t) = z,$	$\forall q \in B_{n-1}, z \in B.$

Such modifications can also be made for bounded versions of other modules.

Next, we illustrate how different types of errors can be handled. Suppose we wish to distinguish the following cases:

- stack empty: operation is illegal because the stack is empty,
- illegal input: operation is illegal because input data is out of bounds,
- stack full: operation is illegal because the stack is full.

We split the illegal state p above into three states: a state, also called p, corresponding to the empty stack violation; state -1, representing all illegal integers; and state 0^{n+1}, representing stack overflow. The modified stack definition is given below. There are no inherent difficulties in handling such error conditions, except for the larger number of cases that need to be distinguished. When an attempt is made to push an illegal integer onto a full stack, we arbitrarily decide to provide the error message "illegal input."

Definition 9. *The* error-handling stack automaton *is a Mealy automaton $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$, where $\Sigma = \{d, p, t\} \cup Z$, $Q = B_n \cup \{p, -1, 0^{n+1}\}$, $q_\epsilon = \epsilon$, $F = B_n$, $\Omega = B \cup \{0, \ldots, n\} \cup \{\text{stack empty, illegal input, stack full}\}$, and*

C1	$\delta(q, z) = qz,$	$\forall q \in B_{n-1}, z \in B,$
C2	$\delta(\epsilon, p) = p,$	
C3	$\delta(\epsilon, -1) = -1,$	
C4	$\delta(0^n, 0) = 0^{n+1},$	
N1	$\delta(q, z) = -1,$	$\forall q \in B_n, z \in Z \setminus B, qz \neq -1,$
N2	$\delta(\epsilon, t) = p,$	
N3	$\delta(q, d) = q,$	$\forall q \in B_n,$
N4	$\delta(p, a) = p,$	$\forall a \in \Sigma,$
N5	$\delta(-1, a) = -1,$	$\forall a \in \Sigma,$
N6	$\delta(0^{n+1}, a) = 0^{n+1},$	$\forall a \in \Sigma,$
N7	$\delta(q, z) = 0^{n+1}$	$\forall q \in B^n, z \in B, qz \neq 0^{n+1},$
N8	$\delta(qz, t) = qz,$	$\forall q \in B_{n-1}, z \in B,$
N9	$\delta(qz, p) = q,$	$\forall q \in B_{n-1}, z \in B,$

O1	$\nu(q,d) = \|q\|$,	$\forall q \in B_n$,
O2	$\nu(qz,t) = z$,	$\forall q \in B_{n-1}, z \in B$,
O3	$\nu(\epsilon,p) =$ stack empty,	
O4	$\nu(\epsilon,t) =$ stack empty,	
O5	$\nu(q,z) =$ illegal input,	$\forall q \in B_n, z \in Z \setminus B$,
O6	$\nu(q,z) =$ stack full,	$\forall q \in B^n, z \in B$.

10 Conclusions

We have developed a theory of trace-assertion specifications which is based on canonical words (traces) and equivalences. It provides a simple formal foundation for the selection of canonical traces, the definition of trace equivalence, and the transformation of traces to canonical form. A key role in our theory is played by prefix-continuous sets of canonical traces. The use of such sets, along with our constructions, guarantees that the process of rewriting any trace to its canonical form is deterministic.

We have shown that the problem of finding equivalence assertions for a module amounts to finding a generating set for its semiautomaton, and we have presented a simple algorithm for finding this set. In contrast to many previous approaches, our method produces the trace equivalence relation completely independently of the concept of legality. Directly from the equivalence assertions, we derive a rewriting system which allows us to transform any trace to its canonical form. This rewriting system has no infinite derivations if and only if the canonical set is prefix-continuous. The set of equivalences is then irredundant. Prefix-continuous sets include both prefix codes and prefix-closed languages as special cases, and can be found with the aid of spanning forests of the semiautomata.

We point out that a specification should use a reduced automaton. The canonical traces are then pairwise observationally inequivalent.

Since canonical traces are representations of the states of the automaton of the module, constructing the trace-assertion specification is equivalent to constructing the automaton. Our results hold for finite and infinite automata.

Finally, we apply our theory to several common modules. These examples illustrate the construction of specifications of modules by automata. If the automata are in standard form, we obtain trace-assertion specifications algorithmically. However, in some case, there are representations of states which are more natural than canonical traces, and, although trace-assertion specifications are always possible, considerable work may be required to obtain them.

Acknowledgments:

We are very grateful to David L. Parnas for his insightful critical comments on earlier versions of this paper, and for a crucial example which challenged us to extend our theory beyond prefix-closed canonical sets; this led us to a complete characterization of the canonical sets for which the rewriting system is well behaved. We also thank John Thistle for an important technical suggestion which led to an improved version of Lemma 1; Mihaela Gheorghiu for carefully reading

several versions of the manuscript and for very useful constructive comments; Jo Atlee and Jan Madey for providing several key references and useful suggestions, and Jack Chen for proposing the example of the set module.

This research was supported by the Natural Sciences and Engineering Research Council of Canada under grants No. OGP0000871 and OGP0000243.

References

1. Bartussek, W. and Parnas, D. L.: Using Assertions About Traces to Write Abstract Specifications for Software Modules. Report No. TR77-012, University of North Carolina at Chapel Hill, December (1977) 26 pp.
2. Bartussek, W. and Parnas, D. L.: Using Assertions About Traces to Write Abstract Specifications for Software Modules. *Inform. Syst. Methodology,* in *Lecture Notes in Computer Science 65*, Springer (1978) 211–236
3. Bartussek, W. and Parnas, D. L.: Using Assertions About Traces to Write Abstract Specifications for Software Modules. Software Fundamentals (Collected Works by D. L. Parnas), D. M. Hoffman and D. M. Weiss, eds., Addison-Wesley (2001) 9–28
4. Book, R. V. and Otto, F.: *String-Rewriting Systems.* Springer-Verlag, Berlin (1993)
5. Brzozowski, J. A. and Jürgensen, H.: Representation of Semiautomata by Canonical Words and Equivalences, pp. 13–27 in Pre-Proceedings, Descriptional Complexity of Formal Systems, 6th Workshop, London, ON, Canada, L. Ilie and D. Wotschke, eds., Report No. 619, Department of Computer Science, University of Western Ontario, London, ON, Canada, July 26–28, 2004. Also at http://maveric.uwaterloo.ca/publication.html
6. Guttag, J. V., Horowitz, E. and Musser, R.: The Design of Data Type Specifications. In *Current Trends in Programming Methodology*, vol. IV, R. T. Yeh, ed., Prentice-Hall, (1978) 60–79
7. Hoffman, D. M.: The Trace Specification of Communications Protocols. IEEE Trans. Computers, vol. C34, no. 12, (1985), 1102–1113
8. Hoffman, D. M. and Snodgrass, R.: Trace Specifications: Methodology and Models. IEEE Trans. Software Engineering, vol. 14, no. 9, (1988), 1243–1252
9. Iglewski, M., Kubica, M., Madey, J., Mincer-Daszkiewicz, J. and Stencel, K.: TAM'97: The Trace Assertion Method of Module Interface Specification. Reference Manual (1997). Technical Report TR 97-01 (238), Institute of Informatics, Warsaw University, Warsaw, Poland, 1997. Also http://w3.uqah.uquebec.ca/iglewski/public_html/TAM/
10. Iglewski, M., Madey, J. and Stencel, K.: On Fundamentals of the Trace Assertion Method. Technical Report TR 94-09 (198), Institute of Informatics, Warsaw University, Warsaw, Poland. September 1994. Also Technical Report RR 94/09-6, Département d'Informatique, Université du Québec à Hull, Hull, QC, Canada, September 1994.
11. Janicki, R. and Sekerinski, E.: Foundations of the Trace Assertion Method of Module Interface Specifications. IEEE Trans. Software Engineering, vol. 27, no. 7, (2001), 577–598
12. Kohavi, Z.: Switching and Finite Automata Theory. McGraw-Hill, New York (1978)
13. Madey, J.: From Requirement Analysis to Code Verification—a Functional Approach (in Polish). Invited talks, 8th Conference on Real-Time Systems, Krynica, September 24–27, 2001, T. Szmuc and R. Klimek (eds.), Chair of Automatics, AGH University of Science and Technology, Cracow, Poland, (2002) 35–74

14. McLean, J.: A Formal Method for the Abstract Specification of Software. J. ACM, vol. 31, no. 3, July (1984), 600–627
15. McLean, J.: Proving Noninterference and Functional Correctness Using Traces, Journal of Computer Security, vol. 1, no. 1 (1992), 37–58
16. McLean, J.: A General Theory of Composition for Trace Sets Closed under Selective Interleaving Functions, Proc. IEEE Symp. on Research in Security and Privacy, Oakland, CA., (1994), 79–93
17. Parnas, D. L. and Wang, Y.: The Trace Assertion Method of Module Interface Specification. Tech. Rept. 89–261, Queen's University, C&IS, Telecommunication Research Institute of Ontario (TRIO), Kingston, ON, Canada (1989)
18. Starke, P. H.: Abstract Automata. North-Holland, Amsterdam (1972)
19. Wang, Y.: Formal and Abstract Software Module Specifications — A Survey. Tech. Rept. 91–307, Computing and Information Science, Queen's University, Kingston, ON, (1991), and Version 2, CRL Report 238, McMaster University, Hamilton, ON, (1994)
20. Wang, Y. and Parnas, D. L.: Simulating the Behavior of Software Modules by Trace Rewriting. IEEE Trans. on Software Engineering, vol. 20, no. 10, October (1994) 750–759
21. Weidenhaupt, K., Pohl, K., Jarke, M. and Haumer, P.: Scenarios in System Development: Current Practice. IEEE Software, vol. 15, no. 2, Mar./Apr. (1998) 34–45

Appendices: Additional Examples

A Queue

This example is based on the queue in [1]. A *queue* is either empty or contains a list $(z_1, \ldots, z_n)$ of integers, where $n > 0$. In the latter case, z_1 is the *front* of the queue and z_n, its *tail*. If $n = 1$, z_1 is both the front and the tail. If the queue is nonempty, operation REMOVE, denoted by r, removes z_1 and the queue now contains $(z_2, \ldots, z_n)$. Also, if the queue is nonempty, operation FRONT, denoted by f, returns z_1 without changing the queue. For each $z \in Z$, operation ADD(z), denoted by z, adds z at the tail of the queue, resulting in $(z_1, \ldots, z_n, z)$. If the queue is empty, r and f are illegal.

We choose $q = z_1 \ldots z_n \in Z^*$ to represent the state of the automaton when the queue contains the list $(z_1, \ldots, z_n)$, and r for the illegal state. The canonical trace for any state is then the corresponding word. The set $Z^* \cup \{r\}$ is prefix-closed.

The queue semiautomaton is illustrated in Fig. 9.

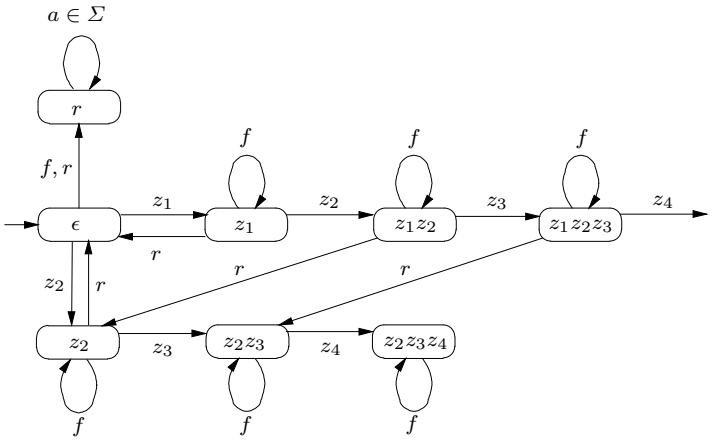

Fig. 9. Queue semiautomaton

Definition 10. *The* queue automaton *is* $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$*, where* $\Sigma = \{f, r\} \cup Z$*,* $Q = Z^* \cup \{r\}$*,* $q_\epsilon = \epsilon$*,* $F = Z^*$*,* $\Omega = Z$*, and* δ *and* ν *are defined below. Note that* $\nu = \nu(q, a)$ *is defined only if* $q \in Z^+$ *and* $a = f$*.*

$$
\begin{array}{lll}
\mathbf{C1} & \delta(q,z) = qz, & \forall q \in Z^*, z \in Z, \\
\mathbf{C2} & \delta(\epsilon, r) = r, & \\
\mathbf{N1} & \delta(\epsilon, f) = r, & \\
\mathbf{N2} & \delta(r,a) = r, & \forall a \in \Sigma, \\
\mathbf{N3} & \delta(zq, f) = zq, & \forall q \in Z^*, z \in Z. \\
\mathbf{N4}: & \delta(zq, r) = q, & \forall q \in Z^*, z \in Z, \\
\mathbf{O1}: & \nu(zq, f) = z, & \forall q \in Z^*, z \in Z.
\end{array}
$$

Proposition 4. *The queue automaton is reduced.*

Proof. State r is rejecting and all the states in Z^* are accepting. Among the accepting states, if $i < j$, then any state q of length i is distinguishable from q' of length j by r^j. Suppose now that q and $q' \neq q$ are of equal length, and their longest common prefix is $q_1 \ldots q_{i-1}$; then $q_i \neq q'_i$. Now q and q' are distinguishable by $r^{n-i}f$. □

B Maximal-Element Module

This example is derived from [1] from the example of the "sorting queue." A *mem* (maximal-element module) is either empty or is a multiset (bag) of integers (duplicates are permitted). If the mem is nonempty, REMOVE, denoted by r, removes one occurrence of the largest integer in the mem. Otherwise, REMOVE is illegal. If the mem is nonempty, MAX, denoted by m, returns the largest integer in the mem without changing it. For each integer $z \in Z$, INSERT(z), denoted by z, inserts z in the mem.

A *multiset* of integers is a mapping $\sigma : Z \to P$ such that, for every $z \in Z$, $\sigma(z)$ denotes the number of occurrences (multiplicity) of z in the multiset. We represent σ as the formal power series

$$\sigma = \ldots + \sigma(-2)x^{-2} + \sigma(-1)x^{-1} + \sigma(0)x^0 + \sigma(1)x^1 + \sigma(2)x^2 + \ldots$$

where x is a new symbol. The *carrier* of σ is the set

$$\text{carrier}(\sigma) = \{x^z \mid \sigma(z) \neq 0\}.$$

A multiset σ is said to be finite or empty, if carrier(σ) is finite or empty, respectively. For multisets, addition is defined component-wise. Subtraction is also component-wise, but is defined only when no co-efficient becomes less than 0.

For a finite, non-empty multiset σ over Z, let

$$\max \sigma = \max \{z \mid x^z \in \text{carrier}(\sigma)\}.$$

Let $\mathbf{0}$ denote the empty multiset, that is,

$$\mathbf{0} = \ldots + 0x^{-2} + 0x^{-1} + 0x^0 + 0x^1 + 0x^2 + \ldots.$$

If $\sigma \neq \mathbf{0}$, r removes a largest element of carrier(σ), resulting in $\sigma - x^{\max \sigma}$, and m returns $\max \sigma$ and leaves σ unchanged. For each $z \in Z$, operation z inserts an additional occurrence of z, resulting in $\sigma + x^z$.

The mem semiautomaton is illustrated in Fig. 10.

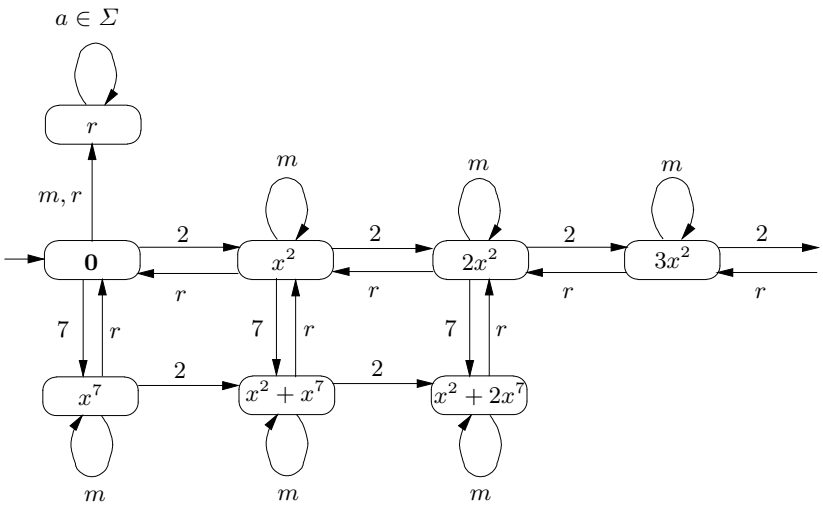

Fig. 10. Mem semiautomaton

Definition 11. *The* mem automaton *is* $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$*, where* $\Sigma = \{m, r\} \cup Z$*,* $Q = Q' \cup \{\infty\}$*,* Q' *is the set of all finite multisets over* Z*,* $q_\epsilon = \mathbf{0}$*,* $F = Q'$*,* $\Omega = Z$*, and*

M1	$\delta(\sigma, z) = x^z + \sigma$,	$\forall \sigma \in Q', z \in Z$,
M2	$\delta(\mathbf{0}, r) = \infty$,	
M3	$\delta(\mathbf{0}, m) = \infty$,	
M4	$\delta(\infty, a) = \infty$,	$\forall a \in \Sigma$,
M5	$\delta(x^z + \sigma, m) = x^z + \sigma$,	$\forall \sigma \in Q', z \in Z$,
M6	$\delta(x^z + \sigma, r) = x^z + \sigma - x^{\max(x^z + \sigma)}$,	$\forall \sigma \in Q', z \in Z$,
O	$\nu(x^z + \sigma, m) = \max(x^z + \sigma)$,	$\forall \sigma \in Q', z \in Z$.

This definition is not in standard form because the states are not represented by traces. We remedy this next. Define function $sort : Z^* \to Z^*$ as follows: $sort(\epsilon) = \epsilon$, and if $w = z_1 \dots z_n$ is any word in Z^+, $sort(w)$ is the word that consists of the integers $z_1, \dots, z_n$ arranged in non-increasing order. For example, $sort(1, 3, 3, 7, 6,) = (7, 6, 3, 3, 1)$. Let $sort(Z^*) = \{sort(z) \mid z \in Z^*\}$.

Legal states are of the form $q \in sort(Z^*)$, and the illegal state is ∞. The natural choice for the canonical trace of q is q itself. Let r be the canonical trace for ∞. The set of canonical traces is prefix-closed. We now construct the automaton from these canonical traces. As before, we denote by $\omega(q)$ the last letter of q if q is nonempty, and ∞ if $q = \epsilon$.

Definition 12. *The* standard mem automaton *is* $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$*, where* $\Sigma = \{m, r\} \cup Z$*,* $Q = sort(Z^*) \cup r$*,* $q_\epsilon = \epsilon$*,* $F = sort(Z^*)$*,* $\Omega = Z$*, and*

C1	$\delta(q,z) = qz,$	$\forall q \in sort(Z^*), z \in Z, z \leq \omega(q),$
C2	$\delta(\epsilon, r) = r,$	
N1	$\delta(q,z) = sort(qz),$	$\forall q \in sort(Z^*), z \in Z, z > \omega(q),$
N2	$\delta(\epsilon, m) = r,$	
N3	$\delta(r,a) = r,$	$\forall a \in \Sigma,$
N4	$\delta(zq, m) = zq,$	$\forall zq \in sort(Z^*), z \in Z.$
N5	$\delta(zq, r) = q,$	$\forall zq \in sort(Z^*), z \in Z,$
O1	$\nu(zq, m) = z,$	$\forall zq \in sort(Z^*), z \in Z.$

This automaton is reduced; the proof is very similar to that for the queue. Moreover, the standard mem automaton is isomorphic to the mem automaton of Definition 11. The standard mem automaton is a particular implementation of the more abstract mem automaton.

This example again illustrates the fact that the "natural" representation of a module by an automaton is not necessarily the one using canonical traces.

C Linked List

This example is similar to the Table/List of [1]. A linked list, which we call *llist*, is initially empty. When nonempty, the llist contains a list of integers and a pointer to the *current* element in the llist. For example, the notation $z_4 z_1 z_3 \dot{z}_1 z_2$ means that the llist now contains $(z_4, z_1, z_3, z_1, z_2)$, and the current pointer points to the fourth element in the list. The INSERT(z) operation, denoted by z, inserts z to the left of the current element, and z becomes the current element. Thus, the new llist is $z_4 z_1 z_3 \dot{z} z_1 z_2$. Operations LEFT and RIGHT, denoted l and r, move the current pointer to the left and right, respectively. Operation DELETE removes the current element and the element to its right becomes current. It is possible to move to the right past the last element in the llist, but not any further.[4] It is not possible to move to the left past the first element. For example, the trace $z_3 z_2 r r z_1 l l d d$ produces the following consecutive llists, starting with the empty llist, ϵ:

$$\epsilon,\ \dot{z}_3,\ \dot{z_2} z_3,\ z_2 \dot{z}_3,\ z_2 z_3,\ z_2 z_3 \dot{z}_1,\ z_2 \dot{z}_3 z_1,\ \dot{z}_2 z_3 z_1,\ \dot{z}_3 z_1,\ \dot{z}_1.$$

In the list $z_2 z_3$ the pointer is just to the right of the last element. Another move to the right is illegal. In $\dot{z}_3$ a move to the left is illegal.

The llist also has operation CURRENT, denoted by c, which returns the value of the current integer, if there is one, and is illegal, otherwise.

For our first definition, in our state representation we use a pair (u, v) of words, and the current pointer is assumed to be on the first letter of v, or to the right of u if $v = \epsilon$.

The llist semiautomaton is illustrated in Fig. 11.

Definition 13. *The* llist automaton *is* $M = (\Sigma, Q', \delta, q'_\epsilon, F', \Omega, \nu')$, *where* $\Sigma = \{c, d, l, r\} \cup Z$, $Q' = (Z^* \times Z^*) \cup \{\infty\}$, $q'_\epsilon = (\epsilon, \epsilon)$, $F' = (Z^* \times Z^*)$, $\Omega = Z$, *and*

[4] In an implementation, one would require another pointer or a doubly linked list. However these issues are not of interest to the specification.

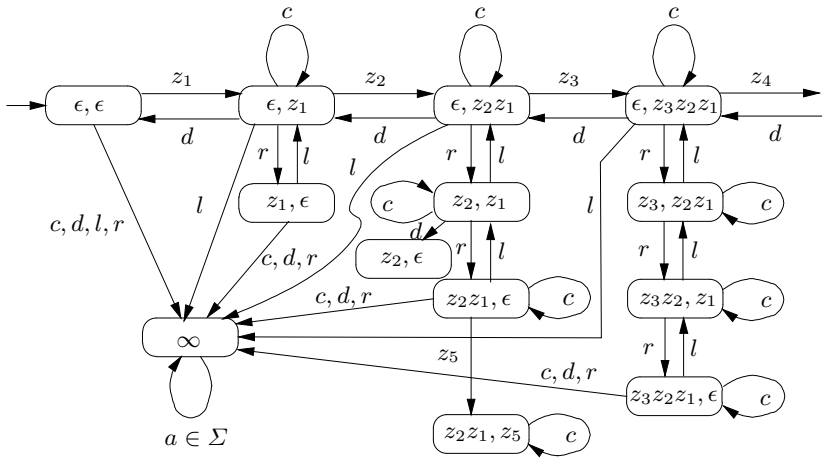

Fig. 11. Llist semiautomaton

M1	$\delta'((u,v),z) = (u,zv)$,	$\forall u,v \in Z^*, z \in Z$,
M2	$\delta'((u,zv),r) = (uz,v)$,	$\forall u,v \in Z^*, z \in Z$,
M3	$\delta'((u,\epsilon),c) = \infty$,	$\forall u \in Z^*$,
M4	$\delta'((u,\epsilon),d) = \infty$,	$\forall u \in Z^*$,
M5	$\delta'((u,\epsilon),r) = \infty$,	$\forall u \in Z^*$,
M6	$\delta'((\epsilon,v),l) = \infty$,	$\forall v \in Z^*$,
M7	$\delta'(\infty,a) = \infty$,	$\forall a \in \Sigma$,
M8	$\delta'((u,zv),d) = (u,v)$,	$\forall u,v \in Z^*, z \in Z$,
M9	$\delta'((uz,v),l) = (u,zv)$,	$\forall u,v \in Z^*, z \in Z$,
M10	$\delta'((u,zv),c) = (u,zv)$,	$\forall u,v \in Z^*, z \in Z$,
O	$\nu'((u,zv),c) = z$,	$\forall u,v \in Z^*, z \in Z$.

While this is a reasonable choice for the state representation, it does not give us a standard automaton because the state representatives are not traces in Σ^*.

For $w \in \Sigma^*$, let w^ρ be the reversal of w. For the canonical trace leading to state (u,v) we choose $(uv)^\rho r^{|u|}$, and we pick c for ∞. This set is prefix-closed. Thus, legal canonical traces are all of the form $w = z_1 \dots z_n r^k$, where $0 \le k \le n$. We introduce the following notation: if $i \le j$, then $w|_i^j = z_i \dots z_j$. Observe that, when $w = z_1 \dots z_n$ is applied, the resulting state is $(\epsilon, z_n \dots z_1)$. If r is now applied n times, the result is $(z_n \dots z_1, \epsilon)$. In any state of the form $(z_n \dots z_1, \epsilon)$, operations c, d, and r are illegal, while l results in $(z_n \dots z_2, z_1)$, and z yields $(z_n \dots z_1, z)$. In case $k < n$, the final state is $(z_n \dots z_{n-k+1}, z_{n-k} \dots z_1)$. Operations c, d, r and z are legal, and l is legal provided $k > 0$. We are now ready to state our standard definition.

Definition 14. *The* standard llist automaton *is* $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$*, where* $\Sigma = \{c,d,l,r\} \cup Z$*,* $F = \{wr^k \mid w \in Z^*, 0 \le k \le |w|\}$*,* $Q = F \cup \{\infty\}$*,* $q_\epsilon = \epsilon$*,* $\Omega = Z$*, and, for* $w = z_1 \dots z_n$*,*

C1	$\delta(w,z) = wz,$	$\forall w \in Z^*, z \in Z$
C2	$\delta(wr^k, r) = wr^{k+1},$	$\forall w \in Z^+, k < n,$
C3	$\delta(\epsilon, c) = c,$	
N1	$\delta(wr^k, z) = w\vert_1^{n-k} z w\vert_{n-k+1}^{n} r^k,$	$\forall w \in Z^*, 0 < k,$
N2	$\delta(wr^{\vert w\vert}, c) = c,$	$\forall w \in Z^+,$
N3	$\delta(wr^{\vert w\vert}, d) = c,$	$\forall w \in Z^*,$
N4	$\delta(wr^{\vert w\vert}, r) = c,$	$\forall w \in Z^*,$
N5	$\delta(w, l) = c,$	$\forall w \in Z^*,$
N6	$\delta(c, a) = c,$	$\forall a \in \Sigma,$
N7	$\delta(wr^k, d) = w\vert_1^{n-k-1} w\vert_{n-k+1}^{n} r^k,$	$\forall w \in Z^+, k < n,$
N8	$\delta(wr^k, l) = wr^{k-1},$	$\forall w \in Z^+, 0 < k,$
N9	$\delta(wr^k, c) = wr^k,$	$\forall w \in Z^+, k < n,$
O1	$\nu(wr^k, c) = z_{n-k},$	$\forall w \in Z^+, k < n.$

One verifies that this automaton is reduced, and isomorphic to the automaton in our first definition.

D Traversing Stack

This example, taken from [8], has some features of both the stack of Section 7 and the linked list of Section C.

A traversing stack, which we call *tstack*, is initially empty. When nonempty, the tstack contains a list of integers and a pointer to the *current* element in the tstack. For example, the notation $z_4 z_1 z_3 \dot{z}_1 z_2$ means that the tstack now contains $(z_4, z_1, z_3, z_1, z_2)$, and the current pointer points to the fourth element in the list. The PUSH(z) operation, denoted by z, is permitted only if either the tstack is empty, or the current pointer points to its leftmost element, which is the top of the tstack. When legal, operation PUSH(z) inserts z to the left of the top element, and z becomes the new top. Operation POP, denoted by p, is legal only if the stack is nonempty and the top element is the current one. Operation RIGHT (called "down" in [8]), denoted r, moves the current pointer to the right, provided there is at least one element to the right of the current one. Operation TOP, legal when the tstack is nonempty, moves the current pointer to the top element. Operation CURRENT, denoted by c, returns the value of the current element.

As in the case of the llist, in our first definition of the tstack we represent each legal state by a pair (u, v) of words. Either both u and v are empty, or $v \neq \epsilon$ and the current pointer is assumed to be on the first letter of v. Let $R = \{\epsilon\} \times Z^+$ and $S = Z^+ \times Z^+$.

The tstack semiautomaton is illustrated in Fig. 12.

Definition 15. *The* tstack automaton *is* $M = (\Sigma, Q', \delta, q'_\epsilon, F', \Omega, \nu')$, *where* $\Sigma = \{c, p, r, t\} \cup Z$, $Q' = R \cup S \cup \{(\epsilon, \epsilon), \infty\}$, $q'_\epsilon = (\epsilon, \epsilon)$, $F' = Q' \setminus \infty$, $\Omega = Z$, *and*

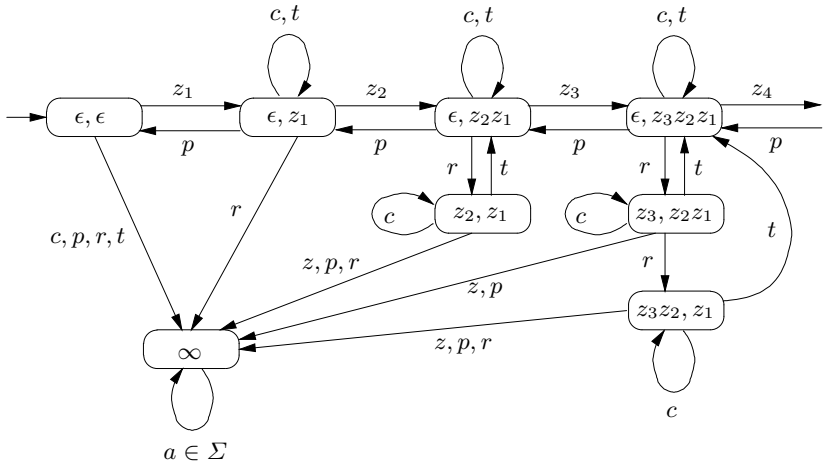

Fig. 12. Tstack semiautomaton

M1	$\delta'((\epsilon, v), z) = (\epsilon, zv)$,	$\forall v \in Z^*, z \in Z$,
M2	$\delta'((u, zz'v), r) = (uz, z'v)$,	$\forall u, v \in Z^*, z, z' \in Z$,
M3	$\delta'((\epsilon, \epsilon), c) = \infty$,	
M4	$\delta'((\epsilon, \epsilon), p) = \infty$,	
M5	$\delta'((\epsilon, \epsilon), r) = \infty$,	
M6	$\delta'((\epsilon, \epsilon), t) = \infty$,	
M7	$\delta'(\infty, a) = \infty$,	$\forall a \in \Sigma$,
M8	$\delta'(q, c) = q$,	$\forall q \in R \cup S$,
M9	$\delta'((\epsilon, zv), p) = (\epsilon, v)$,	$\forall v \in Z^*, z \in Z$,
M10	$\delta'(q, p) = \infty$,	$\forall q \in S$,
M11	$\delta'((u, a), r) = \infty$,	$\forall u \in Z^*, a \in \Sigma$,
M12	$\delta'((u, v), t) = (\epsilon, uv)$,	$\forall u \in Z^*, v \in Z^+$,
M13	$\delta'((u, v), z) = \infty$,	$\forall u, v \in Z^+, z \in Z$,
O	$\nu'((u, zv), c) = z$,	$\forall u, v \in Z^*, z \in Z$.

For the canonical trace leading to state (u, v) we choose $(uv)^\rho r^{|u|}$, and we pick c for ∞. This set is prefix-closed. Thus, legal canonical traces are all of the form $w = z_1 \dots z_n r^k$, where $0 \le k < n$. When $z_1 \dots z_n$ is applied, the resulting state is $(\epsilon, z_n \dots z_1)$. If r is applied $(n-1)$ times, the result is $(z_n \dots z_2, z_1)$. In any such state, operations p, r, and z are illegal, while c does not change the state, and t moves the state back to $(\epsilon, z_n \dots z_1)$. In case $0 < k < n-1$, the final state is $(z_n \dots z_{n-k+1}, z_{n-k} \dots z_1)$. Operations c, r and t are legal, but p and z are illegal. We are now ready to state our standard definition.

Definition 16. *The* standard tstack automaton *is* $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$, *where* $\Sigma = \{c, p, r, t\} \cup Z$, $F = \{wr^k \mid w \in Z^*, 0 \le k < |w|\}$, $Q = F \cup \{\infty\}$, $q_\epsilon = \epsilon$, $\Omega = Z$, *and, for* $w = z_1 \dots z_n$,

C1	$\delta(u, z) = uz,$	$\forall u \in Z^*, z \in Z,$
C2	$\delta(wr^k, r) = wr^{k+1},$	$\forall w \in Z^*, k < n-1,$
C3	$\delta(\epsilon, c) = c,$	
N1	$\delta(\epsilon, p) = c,$	
N2	$\delta(\epsilon, r) = c,$	
N3	$\delta(\epsilon, t) = c,$	
N4	$\delta(c, a) = c,$	$\forall a \in \Sigma,$
N5	$\delta(wr^k, c) = wr^k,$	$\forall w \in Z^+,$
N6	$\delta(wz, p) = w,$	$\forall w \in Z^*, z \in Z,$
N7	$\delta(wr^k, p) = c,$	$\forall w \in Z^+, 0 < k,$
N8	$\delta(wr^k, r) = c,$	$\forall w \in Z^+, k = n-1,$
N9	$\delta(wr^k, t) = w,$	$\forall w \in Z^+,$
N10	$\delta(wr^k, z) = c,$	$\forall w \in Z^+, 0 < k,$
O1	$\nu(wr^k, c) = z_{n-k},$	$\forall w \in Z^+.$

One verifies that this automaton is reduced, and isomorphic to the automaton in our first definition.