Theory of Deterministic Trace-Assertion Specifications

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. Trace assertions are abstract specifications of software modules – "black-box" models of the (finite or infinite) automata representing the modules. Traces are input words, every state is represented by a *canonical trace*, and *trace equivalence* describes the transitions of the automaton. Canonical traces and trace equivalence uniquely determine the automaton. A rewriting system is used to transform any trace to its canonical form. For modules defined by deterministic automata, we present a simple algorithm for trace equivalence and the rewriting system, once a set of canonical traces has been chosen. 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 prove that the rewriting system is always confluent, and that it is Noetherian 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.) We show that each prefix-continuous canonical set corresponds to a spanning forest of the semiautomaton. We derive a complete set of trace assertions directly from the module's automaton. Several examples illustrate our ideas.

1 Introduction

Formal methods are still not universally accepted in the design of commercial software. On the other hand, "scenarios" or "use cases" have become quite popular; see, for example [22]. A scenario usually consists of an English description of a sequence of events specifying a part of the behavior of a software module. For example, a parking ticket machine might satisfy the following scenario: "If a dollar is inserted in the coin slot and then the ticket button is pushed, the machine issues a ticket valid for one hour."

Many software modules can be formally specified by automata. The trace assertion method is based on automata, but 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 hope that this approach will gain wider acceptance than the direct use of automata.

Our work is motivated by a series of papers written by D. 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] (this paper was reprinted in 2001 [3], and a slightly modified version appeared in 1978 [2]). The method has undergone several changes since the original paper: see, for instance, [10–13, 15, 18, 20, 21] for more details and additional references. The main inspiration for our work is the 1994 paper by Wang and Parnas [21]. We generalize, clarify and simplify several concepts presented there.

We provide a theory of deterministic trace-assertion specifications. Trace assertions are abstract specifications of software 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 function calls of the module. The syntax part defines the domains and codomains 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. Every state is represented by a canonical word leading to that state, and trace equivalence describes 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 word leading to a state can be chosen as the canonical word of that state. We show that constructing trace equivalence amounts to finding a set of generators for state equivalence, where two words are state-equivalent if they lead to the same state. We describe a simple algorithm for constructing a set of generators. We prove that, given a set of canonical words and the trace equivalence, one can reconstruct the original automaton uniquely up to isomorphism. This shows that trace-assertion specifications are no more abstract than specifications by automata. To transform any word 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, our rewriting system may have infinite derivations. To remedy this, we impose a condition on the set of canonical words. 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. 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 if and only if the set of canonical words is prefix-continuous.

We demonstrate how to derive a complete set of trace assertions directly from an automaton. Finally, we derive automaton specifications (and hence also trace-assertion specifications) for several modules, such as stacks, queues, linked lists and sets.

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 words are studied in Section 4. Prefix-continuous sets of canonical words are discussed in Section 5. Our theory is illustrated in Section 6 with the simple example of a unary counter. A more complete and more complex example, that of a stack, is given in Section 7, where the "values" section 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. Bartussek and Parnas [1] 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. However, using the approach proposed in [1] and several subsequent papers [2, 15, 18], 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 [15]. 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, that is, 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, any prefix of a legal trace is legal, and only legal traces can return values.

In a 1992 paper [16], 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, and assumes that the empty trace is legal. The right congruence property permits the proofs of equivalence of traces to be more direct. We show that this property is sufficient, that is, the dependence of equivalence on legality is unnecessary.

The interdependence of the definitions of equivalence and legality is removed in the 1994 paper by Wang and Parnas [21] (see also [18]). They propose to identify *canonical traces* as representatives of equivalence classes and a *reduction function* which will transform any trace to its canonical representative. In that paper explicit reference is made to a state machine (deterministic and finite) representing the software module, and four assumptions, missing in the 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 the canonical traces is given in [21] except assumption (1) above. We show below that assumptions (1), (3) and (4) are not necessary.

To prove trace equivalence, [21] uses term rewriting systems. Given a trace, one applies term rewriting rules to it 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.* Wang and Parnas use a heuristic called *smart rewriting* which leads to the canonical trace in many, but not all, cases. 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 call it simply rewriting. We show below 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. The work in [21] is restricted to finite automata, whereas our methods apply to both finite and infinite automata.

After [21], 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.

The work reported in [12] focusses to a large extent on the implementation of the trace assertion method including its syntactic representation. In particular, it provides a comprehensive view of the field as of 1997. In the definition of equivalence, this work deviates from the original proposal of [1] in that two equivalences are considered – a "true" one (called *reduction equivalence*) and an "operational" one (called *behavioural equivalence*); this distinction is needed only because the choice of canonical traces is arbitrary and, therefore, proving trace equivalence may not terminate. In [12], one also has two notions of legality; while this may be useful for applications, it does not add a new feature to the mathematical theory.

In [13], 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 [10]. Timing conditions were not a part of the original proposal in [1]. The work in [10, 11] proposes a heuristic for chosing canonical words. We prove in this paper that this heuristic is indeed appropriate by providing a mathematical foundation for it. In [17], trace assertion methods are used to study security issues in softwares systems.

For a survey of formal specification methods for software modules see [20].

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 free semigroup and the free monoid, respectively, generated by 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 $ua \in X$. Note that 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 $A = (\Sigma, Q, \delta, q_\epsilon)$, 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 $x \in \Sigma^*$,

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

Given any right congruence $\sim$ on Σ^*, we can construct a semiautomaton $S_\sim = (\Sigma, Q_\sim, \delta_\sim, q_\sim)$, as follows. For $w \in \Sigma^*$, let $[w]_\sim$ be the equivalence class of w. Let $Q_\sim$ be the set of equivalence classes of $\sim$, let $q_\sim = [\epsilon]_\sim$, and, for $a \in \Sigma$, let $\delta([w]_\sim, a) = [wa]_\sim$. Note that $S_\sim$ is connected.

It is well-known that the semiautomaton $S_\sim$ is isomorphic to S when $\sim = \equiv_\delta$, with the isomorphism mapping $[w]_\sim$ onto q_w; see [8].

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 they are defined and 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

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

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 the generalized Nerode equivalence is referred to as *observational equivalence*.

For additional material on automata, see, for example, [8, 14, 19].

3.3 Rewriting Systems

In this paper we are concerned with very special rewriting systems. More 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, $w \models^* w'$ if and only if $w = w_0 \models w_1 \models w_2 \models \cdots \models w_n = w'$ for some n, 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 $ux \models vx$, where $u, v \in \Sigma^*$ are specific words and x is an arbitrary word in Σ^*. Systems with this type of rules are known as *regular canonical systems* [5, 6], where "canonical" is a term unrelated to our subsequent usage of the term "canonical." Finite regular canonical systems generate precisely the regular languages and have been studied in detail by Büchi [5, 6]. These systems are equivalent to *expansive* systems in which $|u| \leq |v|$, whereas our systems do not have this property. In fact, in the rewriting systems that we propose for use with trace-assertion specifications, only a finite number of words can be derived from any given word. The second important difference between our work and that of [5, 6] is that we have an infinite number of rules, in general.

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 two important properties:

1. 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 $\tau(w) \models v$.
2. $\models^*$ defines an equivalence $\equiv_{\mathbf{T}}$ as follows: $w \equiv_{\mathbf{T}} w'$ if and only if $\tau(w) = \tau(w')$.

Thus, for an effectively defined confluent Noetherian system, one can compute $\tau(w)$ for every word w, and so decide $\equiv_{\mathbf{T}}$-equivalence of words.

4 Arbitrary Sets of Canonical Words

In this section we make no assumptions about the nature of the set of canonical words. First 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. However, this system may have infinite derivations, a problem which is addressed in Section 5.

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. *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 $\mathbf{X}$.

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*. Note that the pairs are ordered for reasons that will become clear later. 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.

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 $nk - n_0$.*

Proof. Each equation in $\mathbf{G}$ corresponds to a distinct transition of S. There is a total of nk transitions, since there are k transitions out of each state. If u is a canonical word, transitions of the form $\delta(q_u, a) = q_{ua}$, where ua is canonical do not contribute to $\mathbf{G}$. The number of such transitions is n_0. Every transition in which ua is not canonical contributes one equation to $\mathbf{G}$. $\square$

Note that $0 \le n_0 \le n - 1$. If $\mathbf{X}$ is a prefix code, then $n_0 = 0$. At the other extreme, if $\mathbf{X}$ is prefix-closed, then $n_0 = n - 1$.

Lemma 1. $\equiv \subseteq \equiv_\delta$.

Proof. By the construction of $\hat{\mathbf{G}}$, the words in each pair of $\hat{\mathbf{G}}$ lead to the same state, that is, $\hat{\mathbf{G}} \subseteq \equiv_\delta$. By right congruence of $\equiv_\delta$, the claim follows. $\square$

We show later that the converse containment also holds.

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

Lemma 2. *For all $w, w' \in \Sigma^*$, $w \models^* w'$ implies $w \equiv w'$ and therefore $w \equiv_\delta w'$.*

Proof. By definition, each transformation preserves $\equiv$, and $\equiv$ is transitive. By Lemma 1, each transformation also preserves the state. $\square$

Lemma 3. *For $w \in \Sigma^*$, the following hold:*

1. *If no prefix of w is canonical, then $w \models^* w'$ implies $w' = w$.*
2. *If w has a canonical prefix and $w \models^* w'$, then w' has a canonical prefix.*
3. *$w \models^* \chi(q_w)$ if and only if w has a canonical prefix.*

Proof. Suppose no prefix of w is canonical. Then no rule applies to w, because all the rules are of the form $ua \equiv \chi(q_{ua})$, where u is canonical. Consequently, w can only derive itself, and it can do so, because $\models^*$ is reflexive.

For the second claim, suppose w has a canonical prefix. If $w = w'$, the claim holds. If $w \models w'$, then w has the form $w = uav$, where $u, v \in \Sigma^*, a \in \Sigma$, u is canonical and ua is not canonical. Then $w' = \chi(q_{ua})v$, where $\chi(q_{ua})$ is canonical. Now the claim follows by transitivity.

For the third claim, suppose that w has a canonical prefix. We show by induction on the length of w that $w \models^* \chi(q_w)$. If $w = \epsilon$, then w can only have one canonical prefix, namely itself. Thus $\epsilon \models^* \epsilon = \chi(q_\epsilon)$, since $\models^*$ is reflexive; hence the claim holds for the basis case. Now suppose that every word of length less than or equal to n that has a canonical prefix satisfies the claim. Consider $w = ua$ with $|u| = n$ and $a \in \Sigma$, where w has a canonical prefix. If w itself is canonical, then $w \models^* w = \chi(q_w)$. Otherwise, we know that u has a canonical prefix. By the induction assumption, $u \models^* \chi(q_u)$, and so $w = ua \models^* \chi(q_u)a$. If $\chi(q_u)a$ is canonical, then $\chi(q_u)a = \chi(q_{ua}) = \chi(q_w)$, and $w \models^* \chi(q_w)$. Otherwise, $\chi(q_u)a \models \chi(q_{ua})$ is a rule in **T**, and $w = ua \models^* \chi(q_u)a \models \chi(q_{ua})$.

Conversely, if w does not have a canonical prefix, then it can only derive itself. Since w is not canonical, $w \neq \chi(q_w)$. Therefore w cannot derive $\chi(q_w)$. □

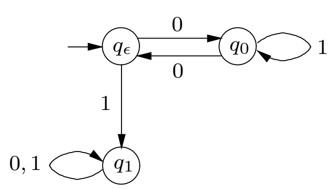

Fig. 1. Semiautomaton S_1

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 $x \in \Sigma^*$:

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 $\chi(q_\epsilon) = 00$; this illustrates Lemma 3 (3). □

Theorem 1. *The rewriting system* **T** *of basic transformations is confluent.*

Proof. Suppose $w \in \Sigma^*$. If w has no canonical prefix, then w can only derive itself, by Lemma 3 (1). Hence w cannot possibly contradict the confluence property. On the other hand, if w does possess a canonical prefix, and $w \models^* w_1$ and $w \models^* w_2$, then w_1 and w_2 also have canonical prefixes, by Lemma 3 (2). By Lemma 3 (3), $w_1 \models^* \chi(q_{w_1})$, and $w_2 \models^* \chi(q_{w_2})$. By Lemma 2, $q_w = q_{w_1} = q_{w_2}$. Hence $w_1 \models^* \chi(q_{w_1}) = \chi(q_w)$, $w_2 \models^* \chi(q_{w_2}) = \chi(q_w)$, and **T** is confluent. □

Definition 2. *Given a set* **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 **W** consists of all the words that do not have a canonical prefix; clearly, **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 **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 **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 2. $\equiv = \equiv_\delta$.

Proof. By Lemma 1, $\equiv \subseteq \equiv_\delta$. 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. If w is acanonical, then ϵ is also 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 **G**, the pair $(\chi(q_w)a, \chi(q_{wa}))$ is in **G**, and our claim follows by transitivity of $\equiv$.

Next, consider a word w in $\mathbf{X}_0 \cup \mathbf{Y}$. By Lemma 3 (3), $w \models^* \chi(q_w)$. By Lemma 2, $w \equiv \chi(q_w)$. This completes the proof. □

It is a disadvantage of the rewriting system **T** that an acanonical word cannot derive its canonical representative. To remedy this, we augment **T** as follows:

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, and to no other word. After this rule is applied, the result is a post-canonical word. By Lemma 3 (2), no acanonical rule is applicable after the first step.

Theorem 3. *Every $w \in \Sigma^*$ derives its canonical representative $\chi(q_w)$ in $\hat{\mathbf{T}}$, and $\hat{\mathbf{T}}$ is confluent.*

Proof. By Lemma 3 (3), the first claim is true for all post-canonical and canonical words. Now consider an acanonical word w. If $w = \epsilon$, then $\epsilon \models \chi(q_\epsilon)\epsilon = \chi(q_\epsilon)$ in $\hat{\mathbf{T}}$. Now suppose that $w \neq \epsilon$. By using the rule $w \models \chi(q_\epsilon)w$, we convert the acanonical word w to the post-canonical word $\chi(q_\epsilon)w$, which then derives in $\mathbf{T}$ the canonical representative $\chi(q_{\chi(q_\epsilon)w})$ of $\chi(q_\epsilon)w$. Thus $w \models^* \chi(q_{\chi(q_\epsilon)w})$ in $\hat{\mathbf{T}}$. Since $\epsilon \equiv \chi(q_\epsilon)$, we have $w \equiv \chi(q_\epsilon)w$, and $q_w = q_{\chi(q_\epsilon)w}$. Hence $\chi(q_w) = \chi(q_{\chi(q_\epsilon)w})$, and so $w \models^* \chi(q_w)$ in $\hat{\mathbf{T}}$.

For the second claim, if a derivation starts with an acanonical word w, only the rule $w \models \chi(q_\epsilon)w$ is applicable. The resulting word $\chi(q_\epsilon)w$ is post-canonical, and only the rules of $\mathbf{T}$ apply to it. In view of Theorem 1, this derivation, like any derivation starting with a post-canonical word, cannot violate confluence. □

Theorem 4. *For any $w, w' \in \Sigma^*$, we have $\chi(q_w) = \chi(q_{w'})$ if and only if $w \equiv w'$.*

Proof. Suppose $\chi(q_w) = \chi(q_{w'})$. Since χ is injective, $q_w = q_{w'}$. By Theorem 2, $w \equiv w'$. Conversely, if $w \equiv w'$, then $q_w = q_{w'}$. Hence $\chi(q_w) = \chi(q_{w'})$. □

One can reconstruct a semiautomaton from its canonical words and equivalences. In fact, 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 $S_{\mathbf{X}} = (\Sigma, \mathbf{X}, \delta_{\mathbf{X}}, \chi(q_\epsilon))$, where, for all $w \in \mathbf{X}, a \in \Sigma$, $\delta_{\mathbf{X}}(w, a) = wa$ if $wa \in \mathbf{X}$, and $\delta(w, a) = \chi(q_{wa})$, if $(wa, \chi(q_{wa})) \in \mathbf{G}$.

Proposition 2. *The semiautomata $S = (\Sigma, Q, \delta, q_\epsilon)$ and $S_{\mathbf{X}} = (\Sigma, \mathbf{X}, \delta_{\mathbf{X}}, \chi(q_\epsilon))$ are isomorphic, with the isomorphism mapping state $q \in Q$ to canonical word $\chi(q) \in \mathbf{X}$.*

Proof. By Theorem 2, the right congruence generated by $\hat{\mathbf{G}}$ is precisely $\equiv_\delta$. □

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. In a specification by canonical words and equivalences one makes a commitment to a particular set of canonical words.

All the results of this section hold for arbitrary canonical sets. Equivalence of two words w and w' is provable in the following sense. By Theorem 3, there exist (finite) derivations $w \models^* \chi(q_w)$ and $w' \models^* \chi(q_{w'})$. By Theorem 4, $w \equiv w'$ if and only if $\chi(q_w) = \chi(q_{w'})$. However, we still have the problem that the rewriting system may permit infinite derivations. This problem is addressed in the next section.

5 Prefix-Continuous Sets of Canonical Words

We now show that, if $\mathbf{X}$ is prefix-continuous, the process of reducing a word to its canonical representative by a derivation in $\hat{\mathbf{T}}$ is deterministic. Equivalence of two words is then proved by reducing them to their canonical representatives, and comparing the representatives. Without prefix-continuity, however, $\hat{\mathbf{T}}$ may allow infinite derivations, as in the next example.

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:

$$\mathbf{T1} \quad 0x \models 01x,\ \mathbf{T2} \quad 10x \models 1x,\ \mathbf{T3} \quad 11x \models 1x,\ \mathbf{T4} \quad 010x \models x,\ \mathbf{T5} \quad 011x \models 01x.$$

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

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

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

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

which never terminates. There is yet another derivation

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

which is also infinite. □

We now overcome the problem of infinite derivations by adding the condition of prefix-continuity.

Lemma 4. *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. If $\mathbf{X}$ is finite, the converse also holds.*

Proof. Suppose there exist words $w, w' \in \mathbf{X}$ and letters $a, a' \in \Sigma$, such that wa and $w'a'$ in $\mathbf{L}$, $wa \neq w'a'$, and wa is a prefix of $w'a'$. Then wa is a prefix of w'. But then, wa must be canonical, since w and w' are canonical, w is a prefix of w', and $\mathbf{X}$ is prefix-continuous. This contradicts the fact that wa is the left-hand side of an equivalence. Hence $\mathbf{L}$ is a prefix code.

Conversely, suppose that $\mathbf{X}$ is finite but not prefix-continuous, and $\mathbf{L}$ is a prefix code. Then there exists $w = uav \in \mathbf{X}$ such that $u \in \mathbf{X}$, and $ua \notin \mathbf{X}$. Consider the infinite set of words $w\Sigma^*$. Since $\mathbf{X}$ is finite, all these words cannot be canonical. Hence there exists some extension wxb of w such that wx is canonical and wxb is not. Therefore $\mathbf{G}$ contains the equivalences $ua \equiv \chi(q_{ua})$ and $uaxb \equiv \chi(q_{uaxb})$, showing that $ua, uaxb \in \mathbf{L}$. Therefore $\mathbf{L}$ cannot be a prefix code. □

The next example shows that the converse of Lemma 4 does not hold in general.

Example 3. In the semiautomaton of Fig. 2, the states are labeled with their canonical representatives. From state 00 on to the right the semiautomaton consists of an infinite binary tree. The set of canonical words is $\{\epsilon, 1\} \cup 00\Sigma^*$, which is not prefix continuous. The set of basic equivalences is $\{0 \equiv 1, 10 \equiv 00, 11 \equiv 00\}$. The set $\mathbf{L} = \{0, 10, 11\}$ of left-hand sides is a prefix code. □

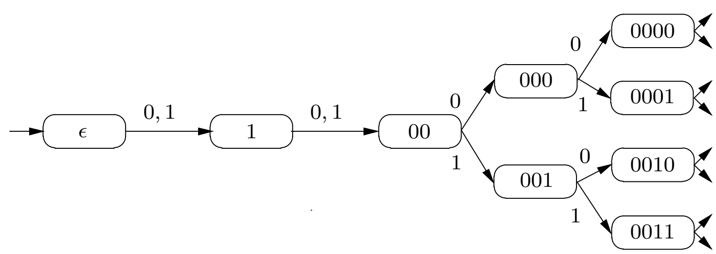

Fig. 2. Semiautomaton illustrating that the converse of Lemma 4 is false

Lemma 5. *At most one rule of $\hat{\mathbf{T}}$ applies to any word if and only if $\mathbf{L}$ is a prefix code.*

Proof. If w is acanonical, the acanonical rule $w \models \chi(q_w)$ is the only rule that applies. If w is minimal canonical, then no rule of $\hat{\mathbf{T}}$ applies to w. If w is post-canonical, then only the rules of $\mathbf{T}$ can be applicable. If $\mathbf{L}$ is a prefix code, at most one rule applies.

Conversely, if $\mathbf{L}$ is not a prefix code, then there exists a post-canonical word to which two rules apply. □

Lemma 6. *If $\mathbf{X}$ is prefix-continuous and $w \in \mathbf{X}$, no rule of $\hat{\mathbf{T}}$ applies to w.*

Proof. As $\mathbf{X}$ is prefix-continuous, w cannot have a canonical prefix u and a non-canonical prefix ua. Hence, by the definition of $\mathbf{T}$, no prefix of w is in $\mathbf{L}$, and no rule applies. Also, no acanonical rule can apply to $w \in \mathbf{X}$. □

Theorem 5. *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 4, $\mathbf{L}$ is a prefix code. By Lemma 5, at most one rule applies to any word. Hence the rewriting process is deterministic. By Theorem 3, each word derives its canonical representative, from which no further derivation is possible, by Lemma 6. Therefore $\hat{\mathbf{T}}$ is Noetherian.

Conversely, suppose that $\mathbf{X}$ is not prefix-continuous. Then there exists $x = uav \in \mathbf{X}$ such that $u \in \mathbf{X}$, but $ua \notin \mathbf{X}$. Therefore $(ua, \chi(q_{ua})) \in \mathbf{G}$, and $x = uav \models \chi(q_{ua})v$. By Lemma 2, x and $\chi(q_{ua})v$ lead to the same state. By Lemma 3 (3), $\chi(q_{ua})v \models^* \chi(q_x) = x$. Thus $x \models \chi(q_{ua})v \models^* x$, and the rewriting system is not Noetherian. □

Theorem 6. *If $\mathbf{X}$ is prefix-continuous, then $\hat{\mathbf{G}}$ is irredundant in the following sense:*

- *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 $\equiv_\delta$.*

Proof. Removing a pair from $\hat{\mathbf{G}}$ is equivalent to removing the corresponding rule from $\hat{\mathbf{T}}$.

If $\epsilon \notin \mathbf{X}$ and $(\epsilon, \chi(q_\epsilon))$ is removed, then the equivalence class of $\equiv$ containing ϵ must be a singleton, since ϵ cannot appear on either side of any rule in $\mathbf{T}$, and the equivalence $\epsilon \equiv \chi(q_\epsilon)$ cannot be derived from any other equivalence by applying the right-congruence property.

Now suppose that $(ua, \chi(q_{ua}))$ is removed from $\mathbf{G}$. By Lemma 6, no rule applies to $\chi(q_{ua})$. On the other hand, ua cannot appear as either side of any other pair in $\mathbf{G}$. By Lemma 5, at most one rule of $\hat{\mathbf{T}}$ applies to any word. Since the only rule applicable to ua has been removed, nothing else is applicable. Hence ua and $\chi(q_{ua})$ must be in different equivalence classes. □

The next example shows that the theorem does not hold in general.

Example 4. Consider the semiautomaton of Fig. 3, where the canonical traces are shown as state labels. Here, $\mathbf{X} = \{\epsilon, 1, 00, 100\}$ is not prefix-continuous. The set of basic equivalences is

$$\{0 \equiv 1, 10 \equiv 00, 11 \equiv 00, 000 \equiv 100, 001 \equiv 100, 1000 \equiv 100, 1001 \equiv 100\}.$$

The equivalence $0 \equiv 1$ implies $000 \equiv 100$ by right congruence. Hence $000 \equiv 100$ is redundant. □

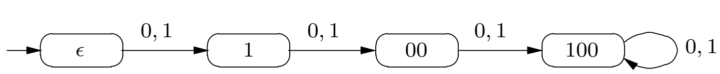

Fig. 3. Semiautomaton with redundant equivalence

Prefix-continuous canonical sets can be found with the aid of certain graph-theoretic concepts. 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 canonical set for S, and it is prefix-continuous.

Example 5. 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 $\{q_\epsilon, q_0, q_1\}$ rooted 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.

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 6. Consider the semiautomaton S_2 of Fig. 4. 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**	$101 \equiv 0$.

Note that all three spanning trees define the same number of basic equivalences, as guaranteed by Proposition 1. □

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

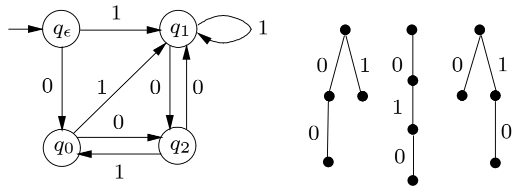

Fig. 4. Semiautomaton S_2 and spanning trees

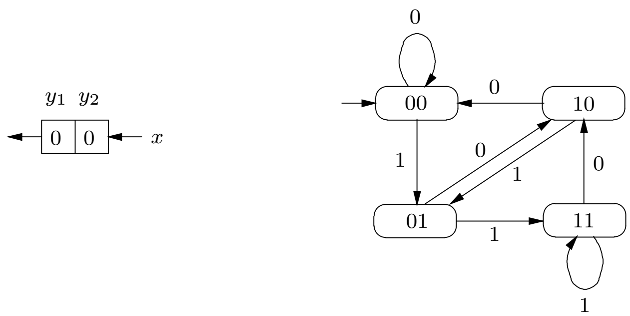

Fig. 5. 2-bit shift register

Example 7. Consider the 2-bit shift register of Fig. 5, 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 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 the eight rewriting rules can be summarized in one statement:

$$abc \equiv bc, \qquad \text{for all } a, b, c \in \Sigma.$$

This symmetry is lost if a prefix-closed set is used. $\square$

6 Unary 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, [21]) 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. A Moore output is a mapping $\mu : Q \to \Theta$, where Θ is some output alphabet. In the remainder of the paper we use only binary Moore outputs, which are normally represented by final and non-final states.

6.1 Counter with Empty Initial State

A unary counter is a pushdown stack, which is initially empty. Only two operations are possible: PUSH and POP. If the stack is empty, POP is illegal and leads to a special illegal state.[1] In any legal state it is possible to PUSH the integer 1 on top of the stack. If the stack contains $(n+1)$ entries, where $n \geq 0$, POP is legal; it removes the top 1 from the stack, leaving n entries. The count is represented by the number of entries on the stack. For convenience, we represent PUSH by 1 and POP, by 0.

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 below.*[2]

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

The state graph of A is shown in Fig. 6 (a), where veritices 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 by an automaton should use a reduced automaton. Otherwise, unnecessary states and transitions are introduced. In a reduced automaton every two distinct states are in different classes of the Nerode equivalence, that is, they are observationally inequivalent. The reader should note, however, that our theory applies equally well to non-reduced automata.

---

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

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

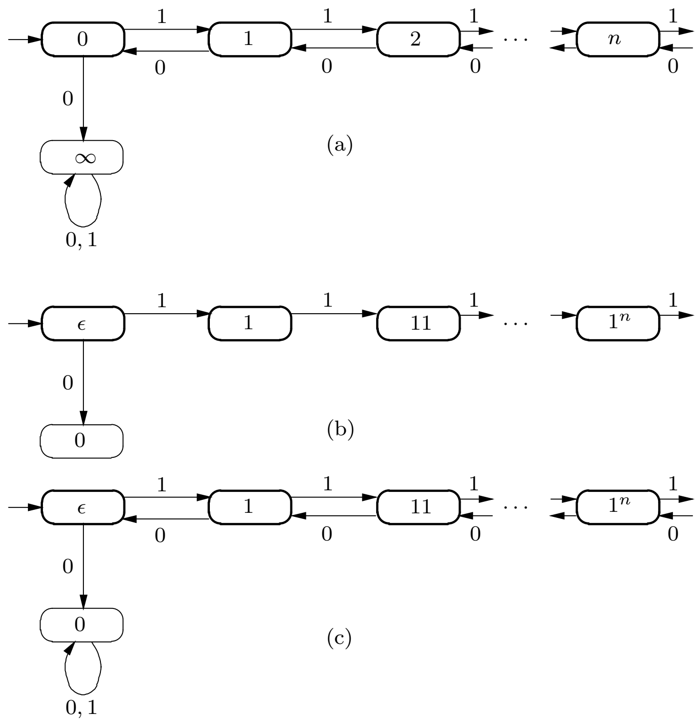

Fig. 6. Counter automaton and canonical words

Proposition 3. *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 $\delta(m, 0^m) = 0 \in F$. □

The first step in constructing a trace-assertion specification is to select canonical words. In the case of our counter, there is only one spanning tree, resulting in canonical word 1^n for the state with n entries, and in 0 for state ∞. Of course, the set $\{1\}^* \cup \{0\}$ is prefix closed. This step is illustrated in Fig. 6 (b).

The second step consists of finding the set **G** of basic equivalences. These equivalences provide the missing transitions in Fig. 6 (b), resulting in Fig. 6 (c).

The basic equivalences and the corresponding basic transformations are

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

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

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

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

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

Now there is a 1-1 correspondence between the **Ni** and the **Ei**. Rules **Ni** correspond to noncanonical extensions of canonical words by letters. Rules **Ci** correspond to canonical extensions of canonical words 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 "$\lambda(w) = \text{true}$" means that w is a legal word. All the canonical words in $\{1\}^*$ are declared legal by **L1** below, and they correspond to the final states of the automaton. The remaining legal words are obtained by the assertion:

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

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

Combining all the parts, we obtain the specification:

Syntax:

$0, 1 : \langle\text{counter}\rangle \rightarrow \langle\text{counter}\rangle$.

Canonical words:

$\{1\}^* \cup \{0\}$

Equivalence:

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

Legality:

$$
\mathbf{L1} \quad \lambda(1^n) = \text{true}, \quad \forall n \in P.
$$

Transformations:

$$
\begin{array}{lll}
\mathbf{T1} & 0ax \models 0x, & \forall a \in \Sigma, x \in \Sigma^* \\
\mathbf{T2} & 1^{n+1}0x \models 1^n x, & \forall n \in P, x \in \Sigma^*.
\end{array}
$$

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, for some reason, we wanted 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. 6 (a), instead of $A = (\Sigma, Q, \delta, 0, F)$, we now use $A = (\Sigma, Q, \delta, 2, F)$, and, for Fig. 6 (c), instead of $A = (\Sigma, \mathbf{X}, \delta_{\mathbf{X}}, \epsilon, A^*)$, we have $A = (\Sigma, \mathbf{X}, \delta_{\mathbf{X}}, 11, A^*)$. In the trace assertion specification, however, we need to find a new spanning forest, and recalculate the equivalences.

In Fig. 7 (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 stack contents. Also, we must calculate a new set of equivalences, in this case:

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

A second solution is shown in Fig. 7 (b), where we use the two trees corresponding to two sets of canonical words: $\{00, 000, 001\}$ and $\{11, 111, \ldots\}$. The advantage of this solution is that, except for three states, the state label denotes the contents of the counter. Now the equivalences are

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

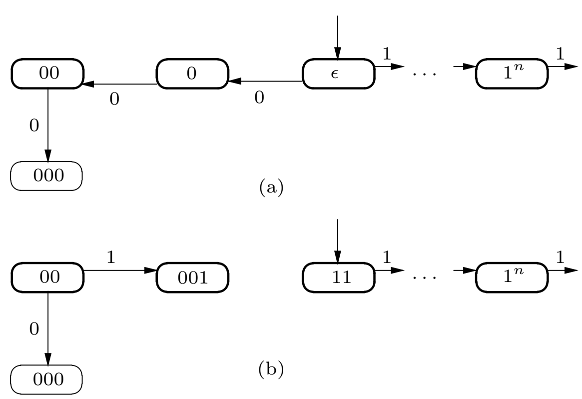

Fig. 7. Counter automaton with changed initial state

To complete the specification, we must add the rule $\epsilon \equiv 11$ to take care of the new initial state. The acanonical words are $\epsilon \cup 0 \cup 1 \cup (01 \cup 10)\Sigma^*$. To find the canonical representative of any nonempty acanonical word w, we use the right congruence property: $\epsilon \equiv 11$ implies $w \equiv 11w$, and then apply the transformation rules from $\mathbf{T}$ to $11w$, which is post-canonical. This approach would require us to test the given word for membership in the set of acanonical words. Alternately, one can put $\chi(q_\epsilon)$ in front of *any* word, and then derive the canonical representative as above, thus avoiding the membership test, at the cost of 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 stack contents $q = z_1 \dots z_n$, with z_n as top, as the representation of a legal state.[3] A natural choice for the canonical word of a state $q \in Z^*$ is q itself. Let p be the canonical word 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* $a = t$.

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

The stack automaton is illustrated in Fig. 8. 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 a few of the transitions, since both Q and Σ are infinite. There is one transition

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

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 if 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.

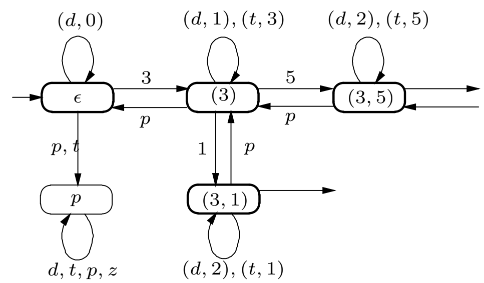

Fig. 8. Stack automaton

Proposition 4. *The stack automaton is reduced.*

Proof. State p is a rejecting state 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 a state q' of length j by the word p^j. Suppose now that q and $q' \neq q$ are of equal length, and their longest common suffix is $q_{i+1} \ldots q_n$; then $q_i \neq q'_i$. Now 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 word is canonical. Hence we examine all the words 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 states 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 words in Z^* are declared legal by **L1** below. The remaining legal words are obtained by **L0**.

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 words, and then make the values applicable to all words by the assertion

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

We now state the complete set of trace assertions for the stack. Each element of Σ results in a state transition. Moreover, inputs d and t produce an output; those inputs map type $\langle\text{stack}\rangle$ into type $\langle\text{stack}\rangle \times \langle\text{integer}\rangle$. Since the syntax assertions are straightforward, we leave them to the reader from now on.

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 $wzpx \models wx, \quad \forall w \in Z^*, z \in Z.$

By construction, this trace-assertion specification of the stack is correct with respect to the stack automaton.

Note: In the rest of our examples in the paper and its appendix we give only the annotated automaton definitions. The interested reader may then easily construct the corresponding trace-assertion specifications. Also, from now on we use generalized Mealy automata.

We include these examples to illustrate the construction of specifications of modules by automata. In this process, we find it very useful to draw partial state graphs for the examples we study. To further simplify the figures, we omit the outputs and show only the transitions of the underlying semiautomata. These 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 [9], discussed also in [20]. 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. 9.

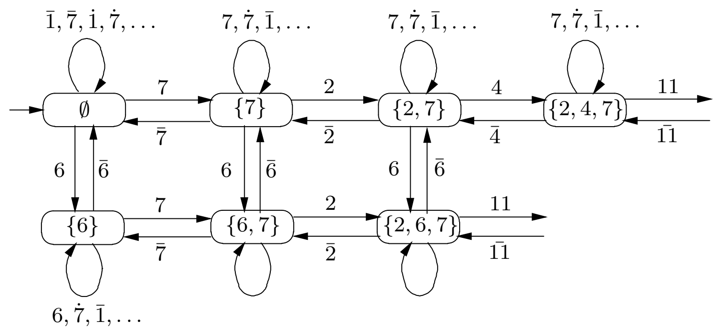

Fig. 9. Set semiautomaton

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 our 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: $\mathit{setsort}(\emptyset) = \epsilon$, and if $q' = \{z_1, \ldots, z_n\} \in Q'$, $\mathit{setsort}(q')$ is the word that consists of $z_1, \ldots, z_n$ arranged in decreasing order. Note that the image $\mathit{setsort}(Q')$ 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 $\mathit{set}(w) = \{z_1, \ldots, z_n\}$. Define function *sort* : $Z^* \to Z^*$ as follows: $\mathit{sort}(\epsilon) = \epsilon$, and if $w = z_1 \ldots z_n$ is any word in Z^+, $\mathit{sort}(w)$ is the word that consists of the integers $z_1, \ldots, z_n$ arranged in non-increasing order. For example, $\mathit{sort}(1, 3, 3, 7, 6,) = (7, 6, 3, 3, 1)$. Let $\mathit{sort}(Z^*) = \{\mathit{sort}(z) \mid z \in Z^*\}$.

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 $Q = \mathit{sort}(Z^*)$. This set is prefix closed. For the canonical word of state $q' \in Q'$, we now choose $\mathit{setsort}(q')$. All words are legal.

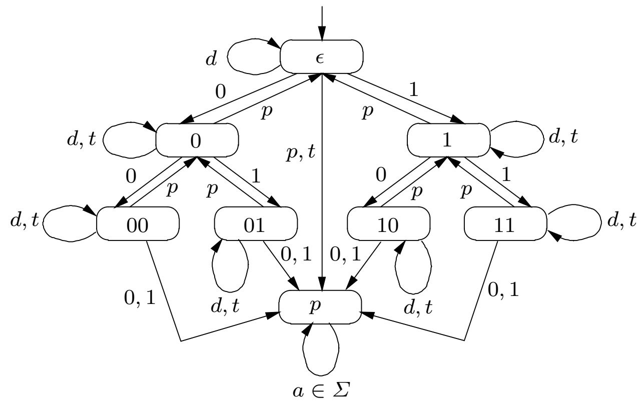

Fig. 10. Bounded stack semiautomaton

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

C1	$\delta(q, z) = \mathit{setsort}(\mathit{set}(q) \cup \{z\}),$	$\forall q \in Q, z \in Z, z \notin q,$
N1	$\delta(q, z) = q,$	$\forall q \in Q, z \in Z, z \in q,$
N2	$\delta(q, \bar{z}) = \mathit{setsort}(\mathit{set}(q) \setminus \{z\}),$	$\forall q \in Q, z \in Z,$
N3	$\delta(q, \dot{z}) = q,$	$\forall q \in Q, z \in Z,$
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 integer is limited to some maximum value b.

Let $B = \{z \mid 0 \le z \le b\}$, 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. 10, with $n = 2$, and $B = \{0, 1\}$.

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

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

It is clear that such simple modifications can also be made in the other modules we have described to handle the bounded cases.

As a second example, we illustrate how different 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* $\Sigma = \{d, p, t\} \cup Z$, $Q = B_n \cup \{p, -1, 0^{n+1}\}$, $q_\epsilon = \epsilon$, $F = B_n$, $\Omega = B \cup \{$ 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	$\delta(\epsilon, z) = -1,$	$\forall z \in Z \setminus B,$
C4	$\delta(q, z) = 0^{n+1},$	$\forall q \in B^n, z \in B,$
N1	$\delta(q, z) = -1,$	$\forall q \in B_n \setminus \{\epsilon\}, z \in Z \setminus B,$
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	$\delta(qz, t) = qz,$	$\forall q \in B_{n-1}, z \in B,$
N8	$\delta(qz, p) = q,$	$\forall q \in B_{n-1}, z \in B,$

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 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.

Our results hold for finite and infinite automata. 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.

Finally, we provide automaton specifications, and hence trace-assertion specifications, for several common modules.

Acknowledgments:

We are very grateful to David 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 thank Mihaela Gheorghiu for carefully reading several versions of the manuscript and for very useful constructive comments, John Thistle for suggesting better versions of Lemma 3 and Theorem 1, Jo Atlee for providing several key references, and Jack Chen for suggesting 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.: 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.: 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.: 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. Büchi, J. R.: Regular Canonical Systems. Archiv für Math. Logik und Grundlagenforschung, **6** (1964) 91–111
6. Büchi, J. R.: *Finite Automata, Their Algebras and Grammars*, (Siefkes, D., ed.), Springer-Verlag (1988)
7. Fülöp, Z., Vágvölgyi, S.: Restricted Ground Tree Transducers. Theoretical Computer Science, **250** (2001) 219–233
8. Gécseg, F. and Peák, I.: Algebraic Theory of Automata. Akadémiai Kiadó, Budapest (1972)
9. 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
10. Hoffman, D.: The Trace Specification of Communications Protocols. IEEE Trans. Computers, vol. C34, no. 12, (1985), 1102–1113
11. Hoffman, D. and Snodgrass, R.: Trace Specifications: Methodology and Models. IEEE Trans. Software Engineering, vol. 14, no. 9, (1988), 1243–1252
12. 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), http://w3.uqah.uquebec.ca/iglewski/public_html/TAM/
13. 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
14. Kohavi, Z.: Switching and Finite Automata Theory. McGraw-Hill, New York (1978)
15. McLean, J.: A Formal Method for the Abstract Specification of Software. J. ACM, vol. 31, no. 3, July (1984), 600–627
16. McLean, J.: Proving Noninterference and Functional Correctness Using Traces, Journal of Computer Security, vol. 1, no. 1 (1992), 37–57
17. McLean, J.: A General Teory of Composition for Trace Sets closed under Selective Interleaving Functions, Traces, Proc. IEEE Symp. on Research in Security and Privacy, Oakland, CA., (1994)
18. 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)
19. Starke, P. H.: Abstract Automata. North-Holland, Amsterdam (1972)
20. Wang, Y.: Formal and Abstract Software Module Specifications — A Survey. Tech. Rept. 91–307, Computing and Information Science, Queen's University, Kingston, ON, (1991)
21. 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
22. 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 from [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 $q \in Z^*$ to represent the state of the automaton when the queue contains the word $q = z_1 \ldots z_n$, and r for the illegal state. The canonical word for any state is then the state itself. The set $Z^* \cup \{r\}$ is prefix-closed.

The queue semiautomaton is illustrated in Fig. 11.

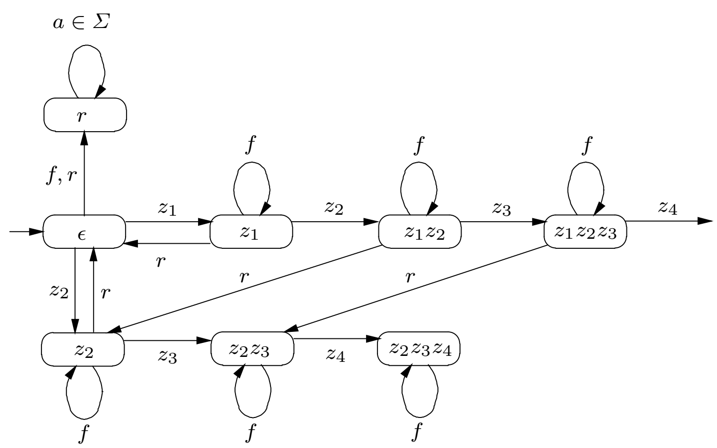

Fig. 11. Queue semiautomaton

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* $a = f$.

C1	$\delta(q, z) = qz,$	$\forall q \in Z^*, z \in Z,$
C2	$\delta(\epsilon, r) = r,$	
N1	$\delta(\epsilon, f) = r,$	
N2	$\delta(r, a) = r,$	$\forall a \in \Sigma,$
N3	$\delta(zq, f) = zq,$	$\forall q \in Z^*, z \in Z.$
N4 :	$\delta(zq, r) = q,$	$\forall q \in Z^*, z \in Z,$

$$\mathbf{O1}: \quad \nu(zq, f) = z, \quad \forall q \in Z^*, z \in Z.$$

Proposition 5. *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. 12.

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*

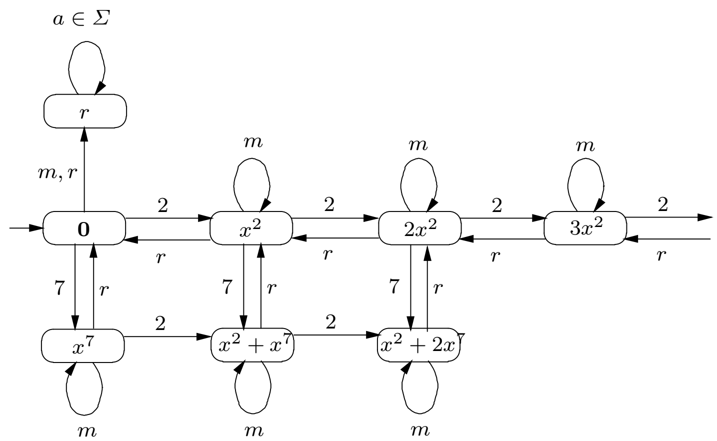

Fig. 12. Mem semiautomaton

M1	$\delta(\sigma, z) = \sigma + x^z,$	$\forall \sigma \in Q', z \in Z,$
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 canonical words. We remedy this next. Define function $sort : Z^* \to Z^*$ as we did for the set module.

Legal states are of the form $q \in sort(Z^*)$, and the illegal state is ∞. The natural choice for the canonical word of q is q itself. Let r be the canonical word for ∞. The set of canonical words is prefix-closed. We now construct the automaton from these canonical words.

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	$\delta(q, z) = sort(qz),$	$\forall q \in sort(Z^*), z \in Z,$
C2	$\delta(\epsilon, r) = r,$	
N1	$\delta(\epsilon, m) = r,$	
N2	$\delta(r, a) = r,$	$\forall a \in \Sigma,$
N3	$\delta(zq, m) = zq,$	$\forall zq \in sort(Z^*), z \in Z.$
N4	$\delta(zq, r) = q,$	$\forall zq \in sort(Z^*), z \in Z,$
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. In fact, the seven parts of each definition are in 1-1 correspondence. It is clear that the standard mem automaton is a particular implementation of the more abstract mem automaton.

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 $z_4z_1z_3\dot{z}_1z_2$ 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 $z_4z_1z_3\dot{z}z_1z_2$. 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 $z_3z_2rrz_1lldd$ produces the following consecutive llists, starting with the empty llist, ϵ:

$$\epsilon,\ \dot{z}_3,\ \dot{z}_2z_3,\ z_2\dot{z}_3,\ z_2z_3,\ z_2z_3\dot{z}_1,\ z_2\dot{z}_3z_1,\ \dot{z}_2z_3z_1,\ \dot{z}_3z_1,\ \dot{z}_1.$$

In the list z_2z_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.

The llist semiautomaton is illustrated in Fig. 13.

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*

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	$\nu'((u, zv), c) = z$,	$\forall u, v \in Z^*, z \in Z$.

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

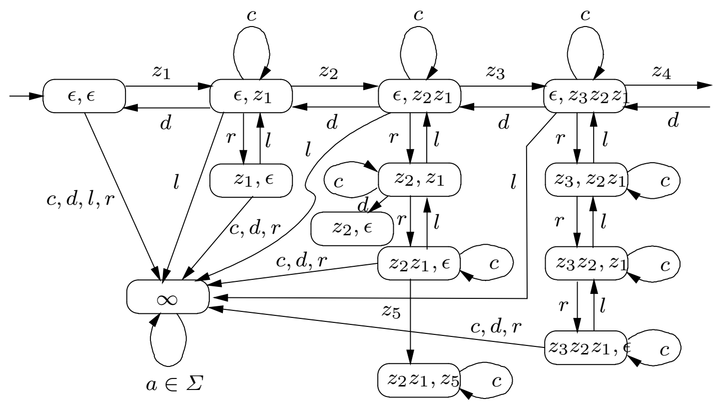

Fig. 13. Llist semiautomaton

While this is a reasonable choice for the state representation, it does not give us a standard automaton because the state representatives are not words 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 $w = z_1 \ldots z_n r^k$, where $0 \le k \le n$. We introduce the following notation: if $i \le j$, then $w|_i^j = z_i \ldots z_j$. Observe that, when $w = z_1 \ldots z_n$ is applied, the resulting state is $(\epsilon, z_n \ldots z_1)$. If r is now applied n times, the result is $(z_n \ldots z_1, \epsilon)$. In any such state, operations c, d, and r are illegal, while l results in $(z_n \ldots z_2, z_1)$, and z yields $(z_n \ldots z_1, z)$. In case $k < n$, the final state is $(z_n \ldots z_{n-k+1}, z_{n-k} \ldots z_1)$. 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* $w = z_1 \ldots z_n$,

C1	$\delta(wr^k, z) = w\|_1^{n-k} z w\|_{n-k+1}^n r^k,$	$\forall w \in Z^*, k \le n = \|w\|,$
C2	$\delta(wr^k, r) = wr^{k+1},$	$\forall w \in Z^+, k < n = \|w\|,$
C3	$\delta(\epsilon, c) = c,$	
N1	$\delta(wr^{\|w\|}, c) = c,$	$\forall w \in Z^+,$
N2	$\delta(wr^{\|w\|}, d) = c,$	$\forall w \in Z^*,$
N3	$\delta(wr^{\|w\|}, r) = c,$	$\forall w \in Z^*,$
N4	$\delta(w, l) = c,$	$\forall w \in Z^*,$
N5	$\delta(c, a) = c,$	$\forall a \in \Sigma,$
N6	$\delta(wr^k, d) = w\|_1^{n-k-1} w\|_{n-k+1}^n r^k,$	$\forall w \in Z^+, k < n = \|w\|,$
N7	$\delta(wr^k, l) = wr^{k-1},$	$\forall w \in Z^+, 0 < k < n = \|w\|,$
N8	$\delta(wr^k, c) = wr^k,$	$\forall w \in Z^+, k < n = \|w\|,$
O1	$\nu(wr^k, c) = z_{n-k},$	$\forall w \in Z^+, k < n = \|w\|.$

Note that the following are corresponding pairs: (**M1, C1**), (**M2, C2**), (**M4, N2**), (**M5, N3**), (**M6, N4**), (**M7, N5**), (**M8, N6**), (**M9, N7**), (**M10, N8**), and (**O, O1**). Rules **C3** and **N1** combined correspond to **M3**. One verifies that this automaton is reduced, and isomorphic to the automaton in our first definition.

D Traversing Stack

This example, taken from [11], 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 [11]), 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. 14.

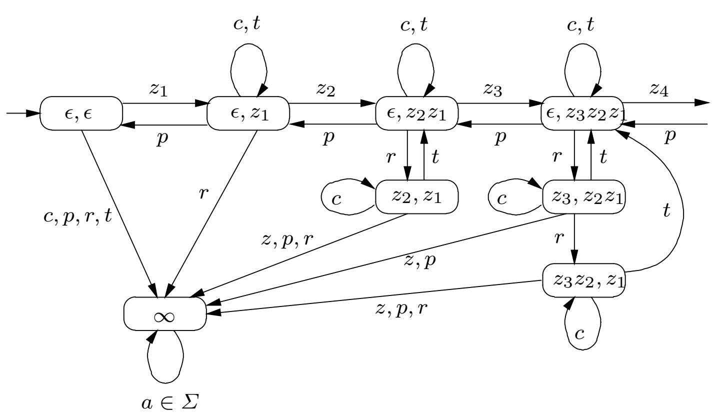

Fig. 14. Tstack semiautomaton

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*

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 $w = z_1 \ldots z_n r^k$, where $0 \le k < n$. When $w = z_1 \ldots z_n$ is applied, the resulting state is $(\epsilon, z_n \ldots z_1)$. If r is applied $(n-1)$ times, the result is $(z_n \ldots z_2, z_1)$. 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 $(\epsilon, z_n \ldots z_1)$. In case $1 < k < n-1$, the final state is $(z_n \ldots z_{n-k+1}, z_{n-k} \ldots z_1)$. 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* $w = z_1 \ldots z_n$,

C1	$\delta(u, z) = uz$,	$\forall u \in Z^*, z \in Z$,
C2	$\delta(wr^k, r) = wr^{k+1}$,	$\forall w \in Z^*, k < |w| - 1$,
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	$\delta(wr^k, c) = wr^k$,	$\forall w \in Z^+, k < |w| - 1$,
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	$\delta(wr^k, r) = c$,	$\forall w \in Z^+, k = |w| - 1$,
N9	$\delta(wr^k, t) = w$,	$\forall w \in Z^+, 0 \le k < |w|$,
N10	$\delta(wr^k, z) = c$,	$\forall w \in Z^+, 0 < k < |w|$,
O1	$\nu(wr^k, c) = z_{n-k}$,	$\forall w \in Z^+, n = |w|$.

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