Trace-Assertion Specifications of Deterministic Software Modules

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 word*, and *trace equivalence* of words describes the transitions of the automaton. Canonical words and trace equivalence uniquely determine the automaton. A rewriting system is used to transform any word to its canonical form.

For modules defined by deterministic automata, we present a simple algorithm for constructing trace equivalence and the rewriting system, once a set of canonical words has been chosen. 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 prove that a set of generators is correct if and only if the empty word is canonical. If it is, the rewriting system is confluent, and it is Noetherian if and only if the set of canonical words is prefix-closed. With prefix-closure, the set of generators is irredundant. Finally, we derive a complete set of trace assertions directly from the module's automaton.

1 Introduction

Our work in this paper was motivated by a series of papers on trace-assertion specifications 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]. The method has undergone several changes since the original paper: see, for instance, [7–10, 12, 13, 15, 16] for more details and additional references.

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.

In essence, a trace-assertion specification consists of five parts: syntax, canonical traces, trace equivalence, legality, and values. Traces are sequences of function calls of the module. The syntax part defines the domains and co-domains of the calls; a canonical trace is a convenient 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 correct from incorrect sequences of calls; the values part defines the output values produced during certain function calls.

In terms of automaton theory, traces are input words. From now on we use the terminology of the theory of automata and languages. Every state is represented by a *canonical word*, and *trace equivalence* of words describes the transitions of the automaton. Given a set of canonical words and the trace equivalence, one can reconstruct the original automaton uniquely. In choosing the canonical words, it is important to be able to transform any word to its canonical form algorithmically. For this purpose, one derives a rewriting system from the trace equivalence. One of the main issues is that the rewriting process should terminate and lead to the equivalent canonical word.

The main inspiration for our work is the 1994 paper by Wang and Parnas [16]. We generalize, clarify and simplify several concepts presented there. To be precise, for modules defined by deterministic finite or infinite automata, we present a simple algorithm for constructing trace equivalence and the rewriting system, once a set of canonical words has been chosen. *A priori*, canonical words can be chosen arbitrarily, but not every choice leads to a correct trace equivalence and well-defined rewriting system. 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 prove that a set of generators is correct if and only if the empty word is canonical. If it is, the rewriting system is confluent, and it is Noetherian if and only if the set of canonical words is prefix-closed. With prefix-closure, the set of generators is irredundant. Finally, we show how to derive a complete set of trace assertions directly from the module's automaton.

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. Generating sets for state equivalence are next studied in Section 4. Transformation of words to canonical form is discussed in Section 5. Section 6 presents a construction of canonical sets of words with desirable properties. These ideas are illustrated in Section 7 with the simple example of a unary counter – an automaton without outputs. A more complete example, that of a stack, is given in Section 8, where the "values" section of the specification is introduced. In Sections 9–14 we present several more challenging examples. Section 15 summarizes our results.

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. As mentioned above, [1] used the concepts of syntax, legality, equivalence, and values. 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, 12, 13], this cannot work in general as the definitions of equivalence and legality depend on each other in such a way that even some obvious equivalences in the case of a stack module cannot be proved.

In 1984, McLean provided a model-theoretic framework for the trace specification method [12]. 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. This, however, does not provide a feasible proof method by which to decide equivalence. Moreover, the key problem, that the definitions of equivalence and legality depend on each other, is still present.

The problem of the interdependence of the definitions of equivalence and legality is partially recognized in the 1994 paper by Wang and Parnas [16] (see also [13]). 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. We show below, why this assumptions together with additional ones missing in [16] are essential for deciding trace equivalence. No specific rule for the choice of the canonical traces is given in [16] except assumption (1) above.

To prove trace equivalence, [16] 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.* As these properties cannot be proved in general for an arbitrary choice of canonical words, Wang and Parnas use a heuristics called *smart trace rewriting* which leads to the canonical word in many, but not all, cases. There are two reasons for this problem as proved below: (1) canonical words – even with the empty word canonical – cannot be chosen in an arbitrary fashion; (2) term rewriting introduces an unmanageable complexity into the problem; this can be avoided by string rewriting over an infinite, but recursively enumerable alphabet. We show below that, if string rewriting is used, it is sufficient that the set of canonical traces be prefix-closed for the rewriting system to be confluent and Noetherian. We also show that a state machine is a more abstract specification than a trace assertion specification. The latter can be derived from the machine and specifies the machine up to isomorphism. Moreover, the specification by a state machine does not require that the machine be finite.

After [16], all publications on the trace assertion method seem to rely on an unexplained choice of the canonical traces. This works because the "natural" or "intuitive" choice usually happens to lead to a provably confluent and Noetherian rewriting system.

The work reported in [9] 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 [9], 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 [10], among other items, the problems of non-deterministic modules and their ramifications are investigated. We do not consider non-deterministic specifications in this paper.

The trace assertion method has also been used for time-dependent systems like communication protocols [7]. Timing conditions were not a part of the original proposal in [1]. The work in [7, 8] proposes a "heuristics" for chosing canonical words. We prove in this paper that these "heuristics" are indeed appropriate.

For a survey of formal specification methods for software modules see [15].

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. A set X is *prefix-closed* if, for any $w \in X$, every prefix of w is also in X.

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, a nonempty set of states, $\delta : Q \times \Sigma \to Q$, the transition function, and $q_\epsilon \in Q$, 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 [5].

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, the output alphabet, and $\nu : Q \times \Sigma \to \Omega$, 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, $L(M) = \{w \mid q_w \in F\}$.

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 $a \in \Sigma$, $\nu'(wa) = \nu(q_w, a)$. Thus $\nu'(wa)$ is defined if and only if $\nu(q_w, a)$ is defined. We write $\nu'(wa) = \nu'(w'a')$ if either both sides are undefined, or both are defined and have the same value. 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 $\equiv_M = \equiv_\delta$.

For additional material on automata, see, for example, [5, 11, 14].

3.3 Rewriting Systems

In this paper we are concerned with a very special type of rewriting system. 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^*.

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 w_c, its *canonical representative,* such that, for any $w_1, w_2 \in \Sigma^*$ with $w \models^* w_1$ and $w \models^* w_2$, one has $w_1 \models^* w_c$ and $w_2 \models^* w_c$ and there is no word $w' \in \Sigma^*$ with $w_c \models w'$.
2. $\models^*$ defines an equivalence $\equiv_{\mathbf{T}}$ as follows: $w \equiv_{\mathbf{T}} w'$ if and only if $w_c = w'_c$.

Thus, for an effectively defined confluent Noetherian system, one can compute the canonical representative of every word and decide equivalence of words.

4 Generating Sets for State-Equivalence

Recall that we are dealing only with connected semiautomata. Our first objective is to find a suitable generating set for the state-equivalence relation of a given semiautomaton. The motivation for this will be given later.

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.* We call the word $\chi(q)$ the *canonical* word of q. Let the set of canonical words be $\mathbf{X}$.

We introduce an equivalence relation $\equiv$ on Σ^*, called the *standard equivalence*, as the smallest right congruence containing the set $\mathbf{G}$ of all pairs of words $(wa, \chi(q_{wa}))$, where $w \in \mathbf{X}$, $a \in \Sigma$, and $wa \notin \mathbf{X}$. We refer to the pairs in $\mathbf{G}$ as *standard generators*. Note that the order in which the elements of a pair are listed is important, for reasons that will become clear soon. Note also that the number of standard generators is infinite in general, although 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 $\mathbf{E1}, \mathbf{E2}, \ldots$

By the construction of $\mathbf{G}$, we always have

$$\equiv \; \subseteq \; \equiv_\delta . \tag{4}$$

However, the converse containment need not hold, as we now show. Consider the semiautomaton S_1 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.

Let $\chi(q_\epsilon) = 00$, $\chi(q_1) = 0$, and $\chi(q_2) = 1$. Then **G** contains the following equivalences: **E1** $01 \equiv 0$, **E2** $10 \equiv 1$, **E3** $11 \equiv 1$, **E4** $000 \equiv 0$, **E5** $001 \equiv 1$. The semiautomaton defined by $\equiv$ has four states corresponding to the equivalence classes of $\epsilon, 0, 1$, and 00; semiautomaton S_1 has only three states. Thus $\equiv_\delta \not\subseteq \equiv$.

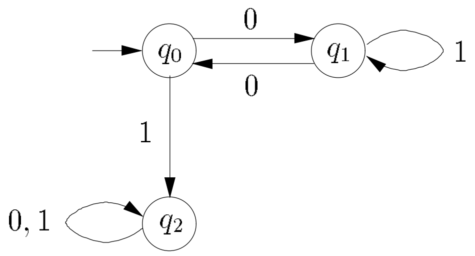

Fig. 1. Semiautomaton S_1

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

For example, consider the semiautomaton of Fig. 1, this time with $\chi(q_\epsilon) = \epsilon$, $\chi(q_1) = 01$, and $\chi(q_2) = 1$. Then we have the following standard equivalences and associated 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** $011x \models 01x$.

Lemma 1. *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 (4), each transformation also preserves the state. □

Lemma 2. *If $\chi(q_\epsilon) = \epsilon$ and $w \in \Sigma^*$, then $w \models^* \chi(q_w)$, and hence $w \equiv \chi(q_w)$.*

Proof. The claim holds for $w = \epsilon$, because $\epsilon = \chi(q_\epsilon)$. Suppose that for all x with $|x| \le n$, $x \models^* \chi(q_x)$ holds, and consider $w = ua$ with $|u| = n$. Now $w \models^* \chi(q_u)a$. If $\chi(q_u)a = \chi(q_{ua})$, then we are done. Otherwise, $(\chi(q_u)a, \chi(q_{ua}))$ is a rule in **G**. So $\chi(q_u)a \models \chi(q_{ua})$, and therefore, $w \models^* \chi(q_w)$. The second statement follows by Lemma 1. □

Theorem 1. *If $\chi(q_\epsilon) = \epsilon$, the rewriting system **T** is confluent.*

Proof. Suppose $w \in \Sigma^*$ is such that $w \models^* w_1$ and $w \models^* w_2$. By Lemma 1, $q_w = q_{w_1}$ and $q_w = q_{w_2}$. Let $x_1 = \chi(q_{w_1})$, and $x_2 = \chi(q_{w_2})$. By Lemma 2, $w_1 \models^* x_1$ and $w_2 \models^* x_2$. Again by Lemma 1, $q_{w_1} = q_{x_1}$ and $q_{w_2} = q_{x_2}$. Thus $q_{x_1} = q_{x_2}$, and it follows that $x_1 = x_2$, since there is only one canonical word per state. Thus $w_1 \models^* x_1$, $w_2 \models^* x_1$, and $\mathbf{T}$ is confluent. □

Theorem 2. $\equiv\, = \,\equiv_\delta$ *if and only if* $\chi(q_\epsilon) = \epsilon$.

Proof. If $\chi(q_\epsilon) = \epsilon$, then $w \equiv \chi(q_w)$ for all $w \in \Sigma^*$, by Lemma 2. Suppose that $w \equiv_\delta w'$; then $q_w = q_{w'}$, and $w \equiv \chi(q_w) = \chi(q_{w'}) \equiv w'$. Thus $\equiv_\delta \subseteq \equiv$. This, together with (4), shows that $\equiv\, = \,\equiv_\delta$. Conversely, suppose $\chi(q_\epsilon) \neq \epsilon$; then $\chi(q_\epsilon) = u$, for some $u \in \Sigma^+$. By the construction of $\mathbf{G}$, ϵ cannot appear on either side of any pair in $\mathbf{G}$. Hence $\epsilon \not\equiv u$, although $\epsilon \equiv_\delta u$. □

Suppose $\chi(q_\epsilon) = \epsilon$. By Lemma 1, $w \models^* w'$ implies $q_w = q_{w'}$, and by Lemma 2, $w \models^* \chi(q_w)$. Hence $q_w = q_{\chi(q_w)}$. Therefore, in our transformation system, every word $w \in \Sigma^*$ has a unique canonical representative $\chi(q_w)$. Define the relation $\equiv_{\mathbf{T}}$ on Σ^* as follows: $w \equiv_{\mathbf{T}} w'$ if and only if $\chi(q_w) = \chi(q_{w'})$. Clearly, this is an equivalence relation. However, one property that holds in Noetherian confluent systems need not hold here: the property that no word can be derived from the canonical word. For example, in the semiautomaton of Fig. 1 with $\chi(q_\epsilon) = \epsilon$, $\chi(q_1) = 01$, and $\chi(q_2) = 1$, the transformation $0x \models 01x$ is applicable to $\chi(q_1)$. This flaw will be corrected in the next section.

Even without the Noetherian property, the following holds.

Theorem 3. *If* $\chi(q_\epsilon) = \epsilon$*, then* $\equiv\, = \,\equiv_{\mathbf{T}}$.

Proof. We will show that $\equiv_\delta \subseteq \equiv_{\mathbf{T}} \subseteq \equiv$. Suppose $w \equiv_\delta w'$; let $q = q_w = q_{w'}$, and let $x = \chi(q_w)$ and $x' = \chi(q_{w'})$. By Lemma 2, $w \models^* x$ and $w' \models^* x'$. By Lemma 1, $q_w = q_x$ and $q_{w'} = q_{x'}$. Thus we have $q_x = q_w = q = q_{w'} = q_{x'}$, showing that $\chi(q_w) = \chi(q_{w'})$. Thus $w \equiv_{\mathbf{T}} w'$, and $\equiv_\delta \subseteq \equiv_{\mathbf{T}}$. By Lemma 2, $\equiv_{\mathbf{T}} \subseteq \equiv$. This concludes the proof, since $\equiv\, = \,\equiv_\delta$ by Theorem 2. □

In summary, for $\mathbf{G}$ to generate $\equiv_\delta$, it is necessary and sufficient to choose the empty word as the canonical word for the initial state; for all other states, the choice is arbitrary.

5 Transformation of Words to Canonical Form

Our second objective is to transform any word w to its canonical representative $\chi(q_w)$. This will be done with the aid of the set $\mathbf{T}$ of standard transformations.

Return to the semiautomaton of Fig. 1, with $\chi(q_\epsilon) = \epsilon$, $\chi(q_1) = 01$, and $\chi(q_2) = 1$. We have the following derivation leading to a canonical word:

$$0001001101 \overset{T1}{\models} 01001001101 \overset{T4}{\models} 01001101 \overset{T4}{\models} 01101 \overset{T5}{\models} 0101 \overset{T4}{\models} 1.$$

Note, however, that we also have the following derivation: $0 \overset{T1}{\models} 01 \overset{T1}{\models} 011 \overset{T1}{\models} 0111 \overset{T1}{\models} \ldots$, which never terminates, and yet another derivation $011 \overset{T5}{\models} 01 \overset{T1}{\models} 011 \overset{T5}{\models} 01 \ldots$, which is also infinite. Thus, in general, the rewriting system defined by the standard transformations may be ill-behaved.

Let $\mathbf{X}$ be the set of all canonical words of a semiautomaton S, and let $\mathbf{L}$ be the set of all left-hand sides of the generating equivalences **Ei**.

Lemma 3. *If* $\mathbf{X}$ *is prefix-closed, then* $\mathbf{L}$ *is a prefix code.*

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

Corollary 1. *If* $\mathbf{X}$ *is prefix-closed, at most one rule applies to any word.*

Note that, if $\mathbf{X}$ is prefix-closed, then $\chi(q_\epsilon) = \epsilon$.

Lemma 4. *If* $\mathbf{X}$ *is prefix-closed and* $w \in \mathbf{X}$*, then no rule in* $\mathbf{T}$ *applies to* w.

Proof. As $\mathbf{X}$ is prefix-closed, every prefix of w is canonical. By the definition of $\mathbf{T}$, no prefix of w is in $\mathbf{L}$. □

Thus, with a prefix-closed canonical set, $\mathbf{T}$ is always well-behaved.

Theorem 4. *If* $\chi(q_\epsilon) = \epsilon$*,* $\mathbf{T}$ *is Noetherian if and only if* $\mathbf{X}$ *is prefix-closed.*

Proof. Suppose that $\mathbf{X}$ is prefix-closed. By Corollary 1, at most one rule applies to any word. By Lemma 2, $w \models^* \chi(q_w)$. By Lemma 4, no rule applies to $\chi(q_w)$. Therefore the rewriting system is Noetherian.

Conversely, suppose $\mathbf{X}$ is not prefix-closed. Let w be canonical, and let u be the shortest prefix of $w = uv$ that is not canonical. By the assumption that ϵ is canonical, we know that $u \in \Sigma^+$. By construction, u appears as a left-hand side of a pair (u, u') in $\mathbf{G}$. Thus $u \equiv u'$ and $w = uv \models u'v$. Note that $u'v$ is not canonical, because uv is canonical, and $q_{uv} = q_{u'v}$. By Lemma 2, $u'v \models^* uv$. Thus $uv \models u'v \models^* uv$, and the rewriting system is not Noetherian. □

6 Prefix-Closed Canonical Sets

It is always possible to find a prefix-closed set of canonical words as follows. Construct a spanning tree of the state graph of the semiautomaton S, with q_ϵ as root, and use the path from the root to a state q as its canonical word $\chi(q)$.

For example, consider the semiautomaton S_2 of Fig. 2. We show three spanning trees for S_2. The corresponding sets of generators are:

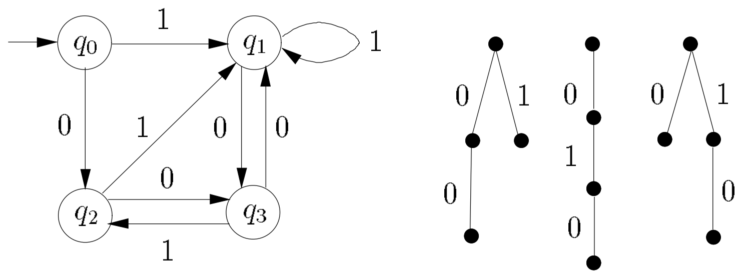

Fig. 2. Semiautomaton S_2 and spanning trees

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** $101 \equiv 0$.

Theorem 5. *If $S = (\Sigma, Q, \delta, q_\epsilon)$ has n states and an alphabet of k letters, there are $nk - (n-1)$ pairs in $\mathbf{G}$, independently of the choice of spanning tree.*

Proof. There are n canonical words, and nk transitions. Every spanning tree uses $n-1$ transitions, since q_ϵ is reached by ϵ, and there are only $n-1$ states to be reached by nonempty words. We claim that, if a transition is in the spanning tree, then it cannot be the left side of a generating pair. For suppose that $\delta(q_\epsilon, w) = q_w$, and $\delta(q_w, a)$ is in the tree. Then $\chi(q_w)$ is canonical, and so is $\chi(q_w)a$. Therefore $\chi(q_w)a$ cannot be the left-hand side of any rule in $\mathbf{G}$.

On the other hand, if $\delta(q_w, a)$ is not in the tree, then $(\chi(q_w)a, \chi(q_{wa}))$ is in $\mathbf{G}$. Consequently, there are $nk - (n-1)$ pairs in $\mathbf{G}$, independently of the choice of spanning tree. $\square$

Theorem 6. *If $\mathbf{X}$ is prefix-closed, the standard set $\mathbf{G}$ of generators is irredundant, that is, if any pair is removed from $\mathbf{G}$, the resulting equivalence is no longer equal to $\equiv_\delta$.*

Proof. Suppose $(wa, w') = (wa, \chi(q_{wa}))$ is removed from $\mathbf{G}$. Then wa cannot appear as either side of any other pair in $\mathbf{G}$. Thus $wa \equiv w'$ must be derived using the right-congruence property of $\equiv$. Hence we must have $wa = uva$, $\chi(q_{wa}) = u'va$, where $u \equiv u'$, for some $u, u', v \in \Sigma^*$. However, since w is canonical, so is every prefix of w; hence no prefix of w can appear as the left-hand side of a pair in $\mathbf{G}$. Similarly, no prefix on w' can appear as the left-hand side of a pair in $\mathbf{G}$, since w' is also canonical. Hence $wa \not\equiv w'$, although $wa \equiv_\delta w'$. Therefore $\mathbf{G} \setminus \{(wa, w')\}$ no longer generates $\equiv_\delta$. $\square$

One can reconstruct a semiautomaton from its canonical words and equivalences. In fact, let $S = (\Sigma, Q, \delta, q_\epsilon)$ be a semiautomaton, and $\mathbf{X}$, a prefix-closed set of canonical words with generating set $\mathbf{G}$. Let $S_{\mathbf{X}} = (\Sigma, \mathbf{X}, \delta_{\mathbf{G}}, \epsilon)$, where, for all $w \in \mathbf{X}, a \in \Sigma$, $\delta(w, a) = wa$ if wa is not the left-hand side of any pair in $\mathbf{G}$, and $\delta(w, a) = w'$, if $(wa, w') \in \mathbf{G}$.

Proposition 1. *The semiautomata* $S = (\Sigma, Q, \delta, q_\epsilon)$ *and* $S_{\mathbf{X}} = (\Sigma, \mathbf{X}, \delta_{\mathbf{G}}, \epsilon)$ *are isomorphic, with the isomorphism mapping state* $q \in Q$ *to canonical word* $\chi(q) \in \mathbf{X}$.

In summary, the information contained in the trace-assertion specification 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 prefix-closed sets of canonical words and the corresponding equivalences as *implementations*, since they are, in fact, less abstract than the semiautomaton.

7 Unary Counter

We now introduce our first example of an infinite semiautomaton, which is also an automaton. We also introduce the concept of legality.

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] 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. In any legal state it is possible to PUSH the integer 1 on top of the stack. The count is represented by the number of entries on the stack. For convenience, we represent PUSH by 1 and POP, by 0.

Definition 1. *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]

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

The state graph of A is shown in Fig. 3, where double circles indicate final states. It should be clear that the automaton corresponds to our informal specification. It should also be clear that a specification of a module by an automaton should use a reduced automaton. Otherwise, unnecessary states and transitions are introduced.

Proposition 2. *The counter automaton is reduced.*

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

Consider the semiautomaton $S = (\Sigma, Q, \delta, q_\epsilon)$ of A. A natural choice for the canonical word of state n is 1^n, and, for ∞, it is 0. The set $\{1\}^* \cup \{0\}$ is prefix closed. The standard equivalences and the corresponding standard transformations are

[1] In general, one could have several illegal states representing various error conditions.
[2] The reason for the particular numbering of items will become apparent later.

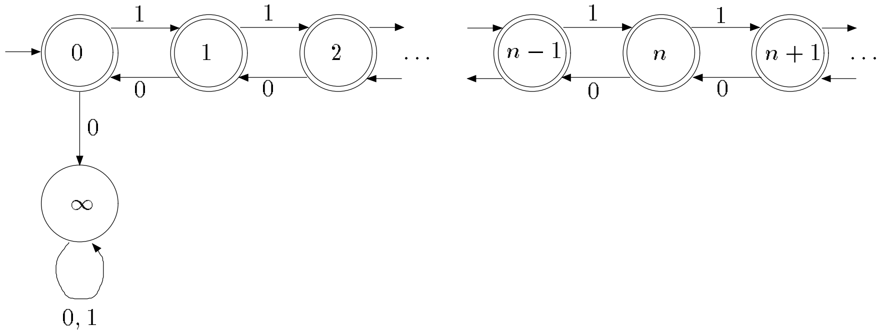

Fig. 3. Counter automaton

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

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

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

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

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

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, and are self-explanatory. For $w \in \Sigma^*$, the assertion "$\lambda(w) = \text{true}$" means that w is a legal word. Since the set of accepting states of A is $F = \{1\}^*$, all the canonical words in $\{1\}^*$ are declared legal by **L1**. The remaining legal words are obtained by the assertion:

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

Syntax:

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

Equivalence:

E1 $0a \equiv 0, \quad \forall a \in \Sigma,$

E2 $1^{n+1}0 \equiv 1^n, \quad \forall n \in P.$

Legality:

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

8 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 $q = z_1 \ldots z_n$, with z_n as top, as the representation of a legal state.[3] The 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 2. *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* $a = t$.

C1 $\delta(q, z) = qz, \quad \forall q \in Z^*, z \in Z,$

C2 $\delta(\epsilon, p) = p,$

N1 $\delta(\epsilon, t) = p$

N2 $\delta(q, d) = q, \quad \forall q \in Z^*,$

N3 $\delta(p, a) = p, \quad \forall a \in \Sigma,$

N4 $\delta(qz, t) = qz, \quad \forall q \in Z^*, z \in Z,$

N5 $\delta(qz, p) = q, \quad \forall q \in Z^*, z \in Z,$

O1 $\nu(q, d) = |q|, \quad \forall q \in Z^*,$

O2 $\nu(qz, t) = z, \quad \forall q \in Z^*, z \in Z.$

The stack automaton is illustrated in Fig. 4. 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 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.

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

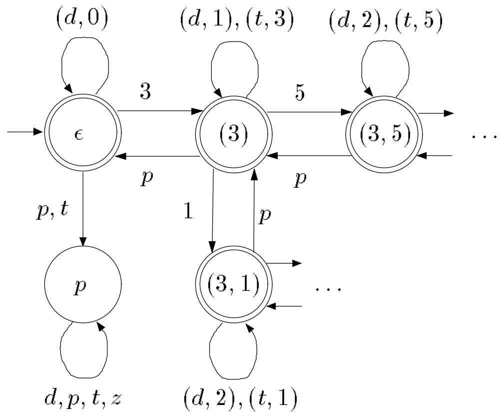

Fig. 4. Stack automaton

Proposition 3. *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 $q_{i+1} \dots q_n$; then $q_i \neq q'_i$. Now q and q' are distinguishable by $p^{n-i}t$. □

The standard 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. By Theorem 2, ϵ must be 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**. 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

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

We now state the complete set of trace assertions for the stack:

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

Equivalence:
E1 $t \equiv p,$
E2 $wd \equiv w,$
E3 $pa \equiv p, \qquad \forall a \in \Sigma,$
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 $\nu(wzt) = z, \quad \forall z \in Z, w \in Z^*.$

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

In the rest of our examples 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. Some of these examples were previously incorrectly specified by trace assertions in the literature.

We find it very useful to draw partial state graphs of the semiautomata we study. They help in deriving the formal definitions and in checking whether all transitions have been considered.

9 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. 5.

Definition 3. *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* $a = f$.

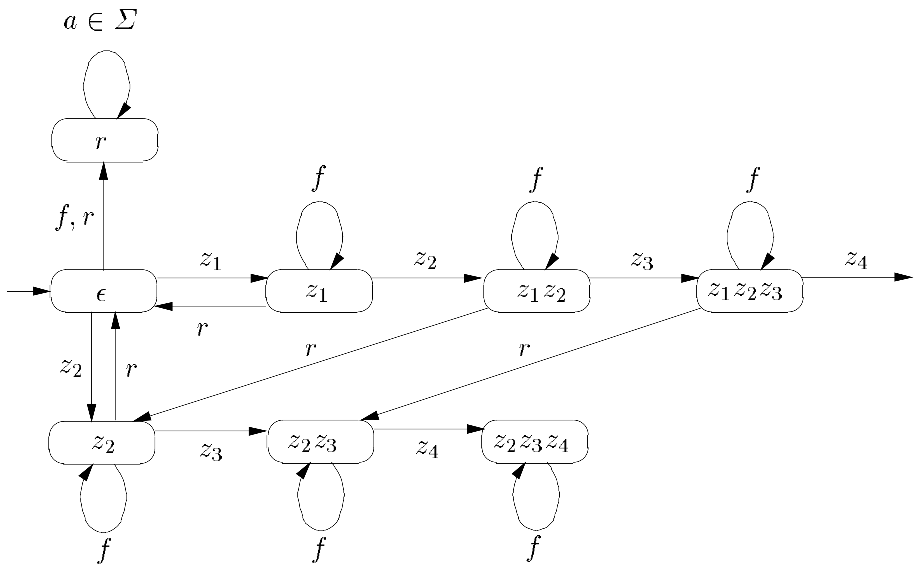

Fig. 5. Queue semiautomaton

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

Proposition 4. *The queue automaton is reduced.*

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

10 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

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

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

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

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. 6.

Definition 4. *The* mem automaton *is* $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$, *where* $\Sigma = \{m, r\} \cup Z$, $Q = Q' \cup \{\infty\}$, Q' *is the set of all finite multisets over* Z, $q_\epsilon = \mathbf{0}$, $F = Q'$, $\Omega = Z$, *and*

M1	$\delta(\sigma, z) = \sigma + x^z,$	$\forall \sigma \in Q', z \in Z,$
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; we remedy this next. Define function $sort : Z^* \to Z^*$ as follows: $sort(\epsilon) = \epsilon$, and if $w = z_1 \ldots z_n$ is any word in Z^+, $sort(w)$ is the word that consists of the integers $z_1, \ldots, z_n$ arranged in non-increasing order. For example, $sort(1, 3, 3, 7, 6,) = (7, 6, 3, 3, 1)$. Let $sort(Z^*) = \{sort(z) \mid z \in Z^*\}$.

Legal states are of the form $q \in sort(Z^*)$, and the illegal state is ∞. The natural choice for the canonical 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.

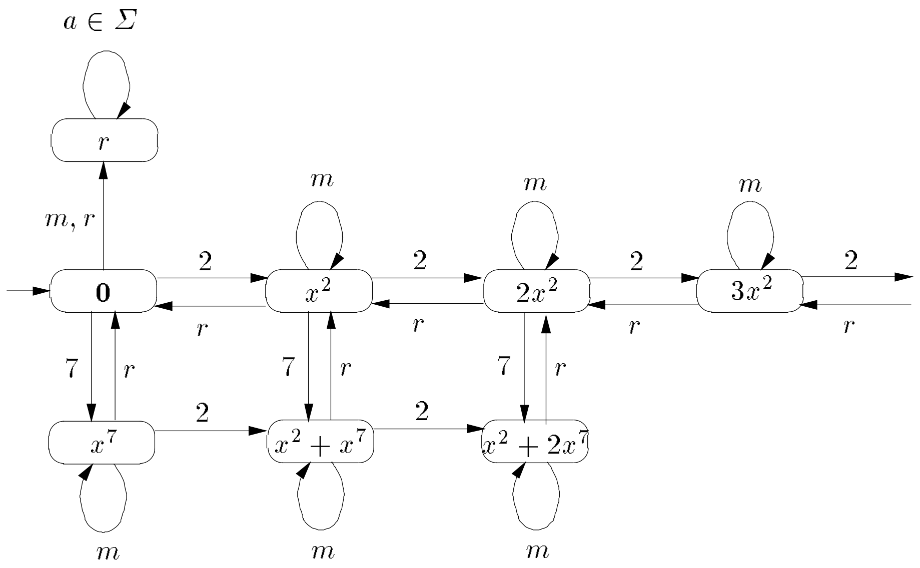

Fig. 6. Mem semiautomaton

Definition 5. *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*

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

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 4. 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.

This example illustrates the fact that the "natural" representation of a module by an automaton is not necessarily the one using canonical words. To get the standard mem automaton, we selected a prefix-closed set of canonical words, and then used the construction of Proposition 1. More examples of this type follow.

11 Set

This example is derived from the"intset" example [6], discussed also in [15]. We start with an empty set S. We can add any integer z to S using INSERT(z), denoted by z; it does not change S if $z \in S$. DELETE(z), denoted by $\bar{z}$, removes z from S, and does nothing if $z \notin S$. MEMBER(z), denoted by $\dot{z}$, returns false if $z \notin S$, and true if $z \in S$.

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

The set semiautomaton is illustrated in Fig. 7.

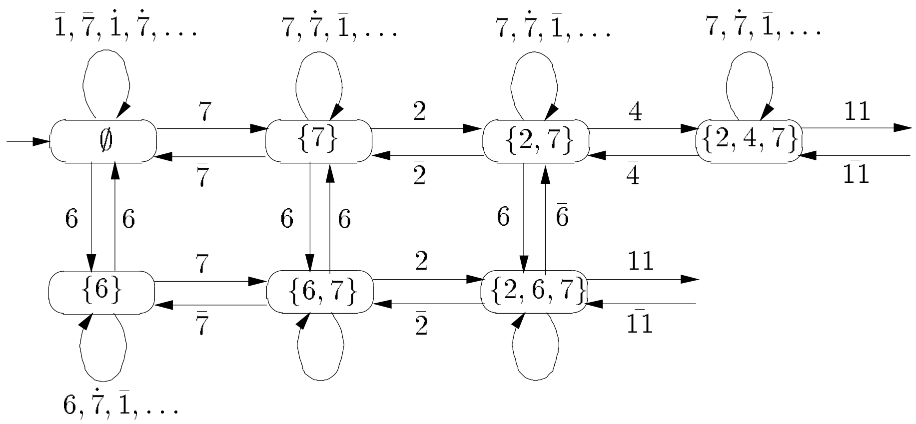

Fig. 7. Set semiautomaton

Definition 6. *The* set automaton *is* $M' = (\Sigma, Q', \delta', q'_0, F', \Omega, \nu')$, *where* $\Sigma = Z \cup \bar{Z} \cup \dot{Z}$, Q' *is the set of all finite subsets of* Z, $q'_0 = \emptyset$, $F' = Q'$, $\Omega = \{\text{true}, \text{false}\}$, *and*

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

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

Define the function *setsort* : $Q' \to Z^*$ as follows: $setsort(\emptyset) = \epsilon$, and if $q' = \{z_1, \ldots, z_n\} \in Q'$, $setsort(q')$ is the word that consists of $z_1, \ldots, z_n$ arranged in decreasing order. Note that the image $setsort(Q')$ is the set of all sorted words without repeated letters. Define function *set* : $Z^* \to Q'$ as follows. If $w = z_1 \ldots z_n \in Z^*$, then $set(w) = \{z_1, \ldots, z_n\}$. Define *sort* as we did for the mem. 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 $Q = sort(Z^*)$. This set is prefix closed. For the canonical word of state $q' \in Q'$, we now choose $setsort(q')$. All words are legal.

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

C1	$\delta(q, z) = setsort(set(q) \cup \{z\}),$	$\forall q \in Q, z \in Z, z \notin q,$
N1	$\delta(q, z) = q,$	$\forall q \in Q, z \in Z, z \in q,$
N2	$\delta(q, \bar{z}) = setsort(set(q) \setminus \{z\}),$	$\forall q \in Q, z \in Z,$
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.

12 Linked List

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

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

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

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

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

The llist semiautomaton is illustrated in Fig. 8.

Definition 8. *The* llist automaton *is* $M = (\Sigma, Q', \delta, q'_0, F', \Omega, \nu')$*, where* $\Sigma = \{c, d, l, r\} \cup Z$*,* $Q' = (Z^* \times Z^*) \cup \{\infty\}$*,* $q'_0 = (\epsilon, \epsilon)$*,* $F' = (Z^* \times Z^*)$*,* $\Omega = Z$*, and*

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

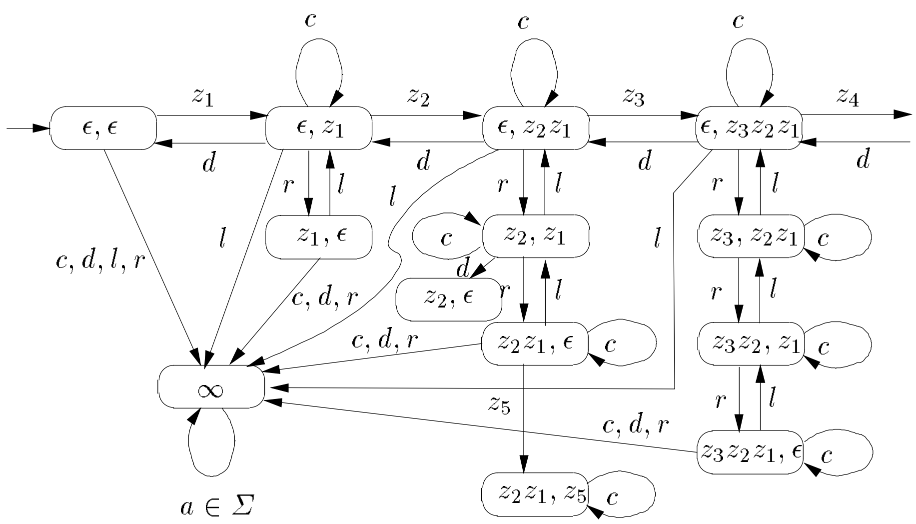

Fig. 8. Llist semiautomaton

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

While this is a reasonable choice for the state representation, it does not give us a standard automaton because the state representatives are not 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 9. *The* standard llist automaton *is* $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$, *where* $\Sigma = \{c, d, l, r\} \cup Z$, $F = \{wr^k \mid w \in Z^*, 0 \leq k \leq |w|\}$, $Q = F \cup \{\infty\}$, $q_\epsilon = \epsilon$, $\Omega = Z$, *and, for* $w = z_1 \ldots z_n$,

C1	$\delta(wr^k, z) = w\|_1^{n-k} z w\|_{n-k+1}^n r^k$,	$\forall w \in Z^*, k \leq n = \|w\|$,
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.

13 Traversing Stack

This example, taken from [8], has some features of both the stack of Section 8 and the linked list of Section 12.

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

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

The tstack semiautomaton is illustrated in Fig. 9.

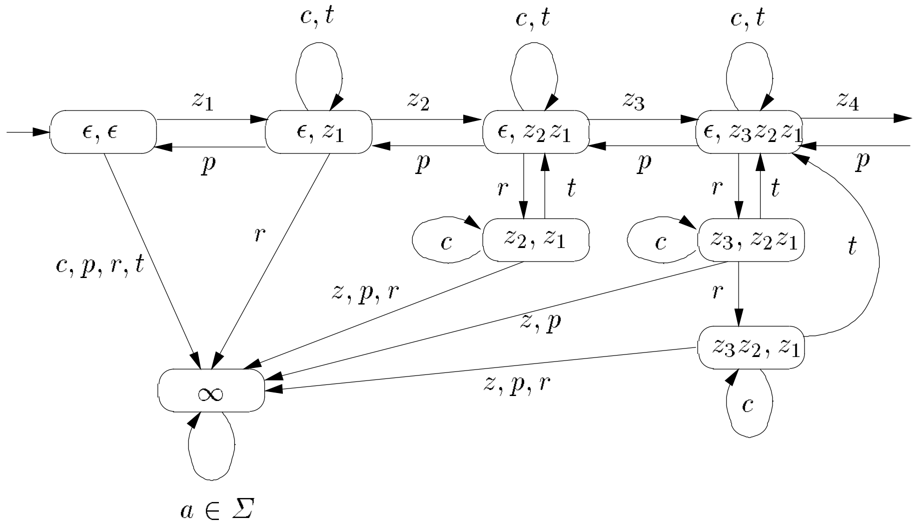

Fig. 9. Tstack semiautomaton

Definition 10. *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 $w = z_1 \dots z_n r^k$, where $0 \leq k < n$. When $w = z_1 \dots z_n$ is applied, the resulting state is $(\epsilon, z_n \dots z_1)$. If r is applied $(n-1)$ times, the result is $(z_n \dots z_2, z_1)$. In any such state, operations p, r, and z are illegal, while c does not change the state, and t moves the state back to $(\epsilon, z_n \dots z_1)$. In case $1 < k < n-1$, the final state is $(z_n \dots z_{n-k+1}, z_{n-k} \dots z_1)$. Operations c, r and t are legal, but p and z are illegal. We are now ready to state our standard definition.

Definition 11. *The* standard tstack automaton *is* $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$, *where* $\Sigma = \{c, p, r, t\} \cup Z$, $F = \{wr^k \mid w \in Z^*, 0 \le k < |w|\}$, $Q = F \cup \{\infty\}$, $q_\epsilon = \epsilon$, $\Omega = Z$, *and, for* $w = z_1 \dots z_n$,

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

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

14 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 8 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 12. *The* bounded stack automaton *is a Mealy automaton* $M = (\Sigma, Q, \delta, q_\epsilon, F, \Omega, \nu)$, *where* $\Sigma = \{d, p, t\} \cup Z$, $Q = B_n \cup \{p\}$, $q_\epsilon = \epsilon$, $F = B_n$, $\Omega = B \cup \{p\}$, *and*

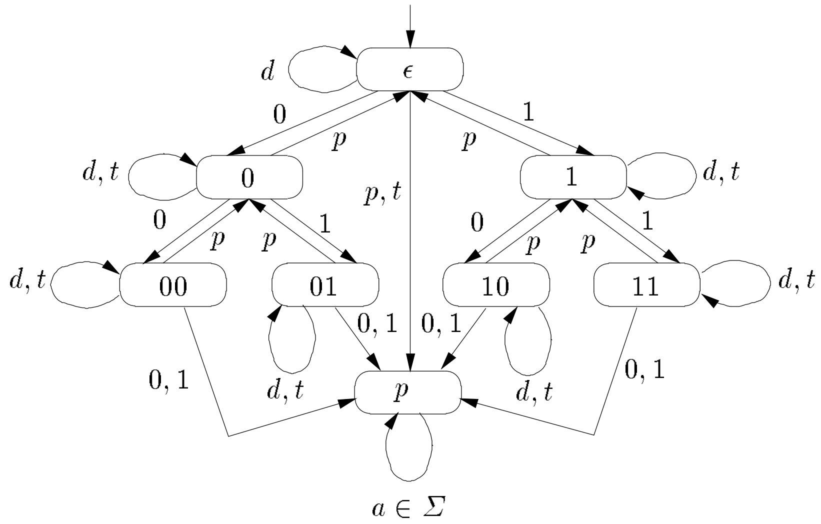

Fig. 10. Bounded stack semiautomaton

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 13. *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	$\nu(q, d) = \lvert q \rvert,$	$\forall q \in B_n,$
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,	$\forall q \in B^n, z \in B.$

15 Conclusions

We have shown that the problem of finding equivalence assertions for a module is equivalent to the problem of finding a generating set for its semiautomaton. We proved that a canonical set of traces generates the state-equivalence if and only if the empty word is canonical. Furthermore, the associated re-writing system is well-behaved if and only if the canonical set is prefix-closed. The canonical sets are then irredundant. We point out that a specification should use a reduced automaton. These results hold for finite and infinite automata. Finally, we provide automaton specifications, and hence trace-assertion specifications, for several common modules.

Acknowledgments: This research was supported by the Natural Sciences and Engineering Research Council of Canada under grants No. OGP0000871 and OGP0000243. The authors thank Jack Chen for suggesting the example of the set module.

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) 111–130

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. Gécseg, F. and Peák, I.: Algebraic Theory of Automata. Akadémiai Kiadó, Budapest (1972)
6. Guttag, J. V., Horowitz, E. and Musser, R.: The Design of Data Type Specifications. In *Current Trends in Programming Methodology*, vol. IV, R. T. Yeh, ed., Prentice-Hall, (1978) 60–79
7. Hoffman, D.: The Trace Specification of Communications Protocols. IEEE Trans. Computers, vol. C34, no. 12, (1985), 1102–1113
8. Hoffman, D. and Snodgrass, R.: Trace Specifications: Methodology and Models. IEEE Trans. Software Engineering, vol. 14, no. 9, (1988), 1243–1252
9. Iglewski, M., Kubica, M., Madey, J., Mincer-Daszkiewicz, J. and Stencel, K.: TAM'97: The Trace Assertion Method of Module Interface Specification. Reference Manual, (1997), http://w3.uqah.uquebec.ca/iglewski/public_html/TAM/
10. 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
11. Kohavi, Z.: Switching and Finite Automata Theory. McGraw-Hill, New York (1978)
12. McLean, J.: A Formal Method for the Abstract Specification of Software. J. ACM, vol. 31, no. 3, July (1984), 600–627
13. 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)
14. Starke, P. H.: Abstract Automata. North-Holland, Amsterdam (1972)
15. Wang, Y.: Formal and Abstract Software Module Specifications — A Survey. Tech. Rept. 91–307, Computing and Information Science, Queen's University, Kingston, ON, (1991)
16. 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