International Journal of Foundations of Computer Science

REPRESENTATION OF SEMIAUTOMATA BY CANONICAL WORDS AND EQUIVALENCES, PART II: SPECIFICATION OF SOFTWARE MODULES

JANUSZ BRZOZOWSKI
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
brzozo@uwaterloo.ca
http://maveric.uwaterloo.ca

and

HELMUT JÜRGENSEN
Department of Computer Science
The University of Western Ontario
London, Ontario, N6A 5B7, Canada
and
Institut für Informatik, Universität Potsdam,
August-Bebel-Str. 89, 14482 Potsdam, Germany

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

A theory of representation of semiautomata by canonical words and equivalences was developed in [7]. That work was motivated by trace-assertion specifications of software modules, but its focus was entirely on the underlying mathematical model. In the present paper we extend that theory to automata with Moore and Mealy outputs, and show how to apply the extended theory to the specification of modules. In particular, we present a unified view of the trace-assertion methodology, as guided by our theory. We illustrate this approach, and some specific issues, using several nontrivial examples. We include a discussion of finite versus infinite modules, methods of error handling, some awkward features of the trace-assertion method, and a comparison to specifications by automata. While specifications by trace assertions and automata are equivalent in power, there are cases where one approach appears to be more natural than the other. We conclude that, for certain types of system modules, formal specification by automata, as opposed to informal state machines, is not only possible, but practical.

Keywords: Automaton, canonical, equivalence, module, rewriting, specification, trace.

2000 Mathematics Subject Classification: 68N30,68Q60,68Q45,68Q42

1. Introduction

Formal methods are still not universally accepted in the design of commercial software. On the other hand, "scenarios" and "use cases" have become quite popular; see, for example [10, 28]. We quote from [10]:

> A scenario is a sequence of steps describing an interaction between a user and a system. A use case is a set of scenarios tied together by a common user goal.

For example, a parking machine might have the following scenario: "If a dollar is inserted in the coin slot and then a button is pushed, the machine issues a receipt valid for one hour."

Our work is motivated by a series of papers written by D. Parnas and his collaborators, and by other authors, over the past 30 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, [12, 13, 15, 16, 20, 23, 26, 27] for more details and additional references. The main inspiration for our work is the 1994 paper by Wang and Parnas [27].

The trace-assertion method is based on automata, but is somewhat similar in nature to the use case approach, and introduces automata rather indirectly. The proponents of trace assertions hope that this approach will gain wider acceptance than the direct use of automata.

A complete trace-assertion specification consists of six parts: syntax, canonical traces, trace equivalence, a rewriting system, legality, and values (outputs). Traces are sequences of function calls of the module. The syntax part defines the domains and co-domains of the functions. A canonical trace is a representative of the set of all traces leading to the same state of the module; it is analogous to a scenario in that it provides a partial description of the automaton. Equivalence identifies the traces leading to the same state and supplies the missing automaton transitions. The rewriting system permits an algorithmic transformation of any trace to its canonical form. Legality distinguishes "normal" from "abnormal" sequences of calls. Finally, the "values" part defines the output values produced by certain function calls.

In [7] we showed that canonical traces, trace equivalence and the rewriting system can be handled conveniently in terms of semiautomata (automata without final states and without outputs). In this paper, we extend the theory to generalized automata, having outputs associated with states (Moore outputs) as well as transitions (Mealy outputs), to obtain complete trace-assertion specifications. The introduction of Moore outputs generalizes the concept of legality and permits the handling of errors. Mealy outputs model the output values of a module.

To illustrate the procedures, we derive automaton specifications (and hence also trace-assertion specifications) for several modules, such as stacks, queues, linked lists and sets. While for some examples the trace-assertion method is quite natural, in others, it is rather awkward. In summary, we generalize, clarify, correct, and simplify several concepts presented in the literature.

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. Our view of the trace-assertion methodology is given in Section 4. Finite versus infinite specifications are discussed in Section 5. Error-handling with the aid of Moore outputs is described in Section 6. In Section 7 we compare the two methods of specifying modules, trace assertions and automata, and illustrate their respective advantages. Section 8 concludes the paper.

The reader is expected to have read [7]; however, we do recall some basic terminology and notation for convenience.

2. Background

The explicit goal of [1] is 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 is 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, 20, 23], 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 provides a model-theoretic framework for the trace specification method [20]. 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 [21], 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.

The interdependence of the definitions of equivalence and legality is removed in the 1994 paper by Wang and Parnas [27] (see also [23]). They propose to identify *canonical traces* as representatives of equivalence classes and a *reduction function* which transforms 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 [27]

except assumption (1) above. To prove trace equivalence, [27] 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. 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. The work in [27] is restricted to finite machines.

The work in [12, 13] proposes a heuristic for chosing canonical words. After [27], all publications on the trace assertion method seem to rely on an unexplained choice of canonical traces.

The work reported in [15] 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 *behavioral equivalence*); this distinction is needed only because the choice of canonical traces is arbitrary and, therefore, proving trace equivalence may not terminate. In [15], 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.

Nondeterministic trace-assertion specifications are first considered in [14], but for more complex, multi-object modules. The nondeterministic model used in [16] is also quite different, because it admits "step sequences", which are sets of words, as traces, and allows input-output pairs as input letters. Neither [14] nor [16] deals with rewriting systems. The method of [7] is extended to nondeterministic semiautomata in [5].

The trace assertion method is also used for time-dependent systems like communication protocols [12]. Timing conditions are not a part of the original proposal in [1]. In [22] trace assertion methods are used to study security issues in softwares systems.

For a 1991 survey of formal specification methods for software modules see [26]; this survey was updated in 1994 [26]. A more recent comprehensive survey, published in 2002 by Madey [19], although focussed on a functional approach to software design, has an extensive bibliography on formal specifications, including trace-assertion methods.

The work in [5, 6, 7], together with the present paper, provides a mathematical foundation for much of the trace-assertion methodology, and clarifies most of the issues raised in the earlier literature. Specifically, the main features of this work are as follows:

1. Modules representable by both finite and infinite automata are handled.

2. Both deterministic and nondeterministic specifications are modeled.

3. A prescription (prefix-continuity), that guarantees a well-behaved model, is given for choosing sets of canonical words, thus eliminating the need for heuristics. The empty trace need not be canonical.

4. In the deterministic case, to find a set of prefix-continuous canonical words one can use any spanning forest of the automaton.

5. Trace equivalence and legality are independent of each other.

6. A generating set for trace equivalence is derived algorithmically from an automaton.

7. A string rewriting system is constructed directly from the equivalence generating set.

8. In the deterministic case, any trace can be rewritten in at most one way. The rewriting system is always confluent and Noetherian, and the canonical representative of any trace is obtained algorithmically.

9. In the nondeterministic case, any trace can be rewritten in finitely many ways. Moreover, at most one prefix of the trace can be rewritten in a step, though several rewriting rules may apply. The rewriting system is always Noetherian, and all canonical representatives of a trace are obtained algorithmically.

10. The concept of legality is replaced by a classification of canonical traces into several types. This facilitates the handling of error conditions.

11. Outputs are defined for canonical traces, and extended to other traces using trace equivalence.

12. Each trace-assertion specification determines a unique automaton, and *vice versa*.

13. The concept of observational equivalence of traces corresponds to a generalized Nerode equivalence. The automaton is reduced if no two states are Nerode-equivalent.

14. Normally, two canonical traces should correspond to two different behaviors of the module. However, even if the equivalence of two canonical traces is not recognized, the specification is still correct, though not minimal.

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 *prefix-free* 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. 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-free and prefix-closed sets are prefix-continuous.

3.1. Semiautomata and Equivalences

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

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

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

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

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

3.2. Automata with Markings and Outputs

For additional material on automata, see, for example, [17, 25].

Given a semiautomaton $S = (\Sigma, Q, \delta, q_\epsilon)$ one can add two types of outputs to it, Moore and Mealy.

A *deterministic Moore automaton* is a tuple $M_\mu = (\Sigma, Q, \delta, q_\epsilon, \Xi, \mu)$, where $(\Sigma, Q, \delta, q_\epsilon)$ is a semiautomaton, Ξ is a nonempty *mark alphabet,* and $\mu : Q \to \Xi$ is the *marking function.* Each input word w defines the mark $\mu(q_w)$. The function $\mu : Q \to \Xi$ uniquely determines a function $\mu' : \Sigma^* \to \Xi$ as follows: For $w \in \Sigma^*$, $\mu'(w) = \mu(q_w)$. In the sequel, we refer to μ' simply as μ. A Moore automaton is an *acceptor* if Ξ has one or two elements.

A *deterministic Mealy automaton* is a tuple $M_\nu = (\Sigma, Q, \delta, q_\epsilon, \Omega, \nu)$, where $(\Sigma, Q, \delta, q_\epsilon)$ is a semiautomaton, Ω is the *output alphabet,* and $\nu : Q \times \Sigma \to \Omega$ is a partial function, the *output function.* 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.

Combining the outputs of Moore and Mealy automata, we obtain a *deterministic Moore-Mealy automaton*, or simply *automaton*, which is a tuple $M = (\Sigma, Q, \delta, q_\epsilon, \Xi, \mu, \Omega, \nu)$, where $M_\mu = (\Sigma, Q, \delta, q_\epsilon, \Xi, \mu)$ is a Moore automaton and $M_\nu = (\Sigma, Q, \delta, q_\epsilon, \Omega, \nu)$ is a Mealy automaton.

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

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

Note that the following always holds: $w \equiv_\delta w' \Longrightarrow w \equiv_M w'$. An automaton M is *reduced* with respect to the equivalence $\equiv_M$ if and only if $w \equiv_M w' \Longrightarrow w \equiv_\delta w'$. 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*.

3.3. Prefix-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). Let $\mathbf{T} \subseteq \Sigma^* \times \Sigma^*$ be a binary relation on Σ^*. The pairs in $\mathbf{T}$ are called *rewriting rules*, and $\mathbf{T}$ is a *rewriting system.* Given any $w, w' \in \Sigma^*$, we say that w *rewrites* to w', written $w \models w'$, if there is some $(y, v) \in \mathbf{T}$ such that $w = yx$ and $w' = vx$. We say then that rule (y, v) *applies* to w.

Systems with this type of rules are known as *regular canonical systems* [8, 9], 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 [8, 9]. However, we are not concerned with languages, but only with semiautomata. Moreover, we have an infinite number of rules and deal with infinite semiautomata, in general. These systems are also called *prefix-rewriting systems* [18], and can be viewed as ground-term-rewriting systems [24].

The reflexive and transitive closure of $\models$ is denoted by $\models^*$. 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 case w derives w' in n steps, we also write $w \models^n w'$; note that $w \models^0 w'$ if and only if $w = w'$.

A word $w \in \Sigma^*$ is *irreducible by* $\mathbf{T}$ (or simply *irreducible,* if $\mathbf{T}$ is understood), if there is no $w' \in \Sigma^*$, such that $w \models w'$, that is, if no rule applies to w.

A rewriting system is *Noetherian* if there is no word w from which a derivation of infinite length exists. It 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 $w_2 \models^* w'$. 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 $\tau(w)$ is irreducible by $\mathbf{T}$.

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. The Trace-Assertion Methodology

Our view of the trace-assertion methodology follows. We assume that the module to be specified is deterministic.

Input Alphabet: Identify the function calls to the module together with the parameter values. In *our* version of the methodology, an *input* is a pair `(function, value)`. For example, suppose in a stack the function `push` can accept three parameter values 1, 2 and 3. Instead of the usual single operation `push(item)` with one parameter taking three possible values, we introduce inputs `(push,1)`, `(push,2)` and `(push,3)`. To simplify the notation, we write `push1`, `push2` and `push3`.

Output Alphabet: Identify the outputs of the module. For example, the function call `top` to a nonempty stack returns the value of the top element.

Syntax: Specify the data types of the module. For example, a stack, like the one above, would require such declarations as:

$$\begin{aligned}
\texttt{push1} &: \langle\texttt{stack}\rangle \to \langle\texttt{stack}\rangle \\
\texttt{push2} &: \langle\texttt{stack}\rangle \to \langle\texttt{stack}\rangle \\
\texttt{push3} &: \langle\texttt{stack}\rangle \to \langle\texttt{stack}\rangle \\
\texttt{pop} &: \langle\texttt{stack}\rangle \to \langle\texttt{stack}\rangle \\
\texttt{top} &: \langle\texttt{stack}\rangle \to \langle\texttt{stack}\rangle \times (\langle\texttt{integer}\rangle \cup \{\epsilon\}) \\
\texttt{depth} &: \langle\texttt{stack}\rangle \to \langle\texttt{stack}\rangle \times (\langle\texttt{integer}\rangle \cup \{\epsilon\})
\end{aligned}$$

In this stack, the first four inputs do not produce any outputs, whereas `top` and `depth` may produce either an integer output or ϵ, which denotes "no output".

Every input to a module results in a state transition; this includes transitions in which the state does not change. Therefore, a declaration has one of these two forms:

$$\begin{aligned}
\texttt{input1} &: \langle\texttt{state}\rangle \to \langle\texttt{state}\rangle \\
\texttt{input2} &: \langle\texttt{state}\rangle \to \langle\texttt{state}\rangle \times (\langle\texttt{output}\rangle \cup \{\epsilon\})
\end{aligned}$$

Canonical Traces: Select a set **X** of canonical traces for the module. Together, these traces represent the most typical uses of the module. Normally, two distinct canonical traces should correspond to two different behaviors of the module. The difference might not be observable immediately after the two traces are applied, but there should exist a continuation by which they become distinguishable. Even if the equivalence of two canonical traces is not recognized, a correct specification is still derived, though it is not minimal.

Typically, one starts with the empty trace, meaning that nothing has been done yet, and continues by extending the set of canonical traces obtained so far by single inputs. This approach naturally leads to a prefix-continuous set of canonical traces which includes the empty trace. There may, however, be valid reasons not to start with the empty trace, as illustrated by the shift-register example in [7]. It is also conceivable that one may come up with

a "natural" set of canonical traces which is not prefix-continuous. This still leads to a valid specification of the module, although the rewriting system is ill-behaved. A remedy to this problem is discussed below.

Trace Equivalence Construct the set $\mathbf{G}$ of basic equivalences as described in [7]. Now one has a complete description of the underlying semiautomaton of the module. If $\mathbf{X}$ is not prefix-continuous, one can derive a new prefix-continuous set of canonical traces from this semiautomaton with the aid of any spanning forest as in [7], and construct a new set $\mathbf{G}$.

If the empty trace is not canonical, the pair $(\epsilon, \chi(q_\epsilon))$ needs to be added to $\mathbf{G}$ to obtain $\hat{\mathbf{G}}$, the generating set for the trace-equivalence relation, where $\chi(q_\epsilon)$ is the canonical trace of the initial state. Otherwise, $\hat{\mathbf{G}} = \mathbf{G}$.

Rewriting System Directly from $\hat{\mathbf{G}}$ generate the rewriting system $\hat{\mathbf{T}}$. Recall that any trace which does not have a canonical trace as a prefix is called *acanonical*. For each such trace w, one introduces the acanonical rule $w \models \chi(q_\epsilon)w$. This system is guaranteed to be confluent and Noetherian.

Legality Issues Some canonical traces may lead to errors. It may be convenient to classify the canonical traces according to error types. For example, consider an empty stack. The input `push1` is a correct operation (type 0, say), whereas `pop` is an error of type 1, and `top` is an error of type 2. This distinction of behavior types can be handled by Moore outputs of the automaton of the module. This generalizes the concept of legality introduced in [1], where only two behavior types are used. Note that this part of the specification is optional.

Outputs Outputs are modeled as Mealy outputs of the automaton of the module. For each canonical trace w and each output-producing input a, assign the corresponding output value for trace wa. A trace u that is not canonical inherits the output for ua through the equivalence relation, as follows. Using the (deterministic) rewriting system $\hat{\mathbf{T}}$, find the canonical representative w of u, and look up the output for wa.

Reduced Specifications As mentioned, some canonical traces could be equivalent, that is, they could result in the same behavior of the module. In that case, the automaton constructed from the trace-assertion specification has equivalent states with respect to the generalized Nerode equivalence. One can obtain a more compact specification from a spanning forest of the reduced automaton.

Example 1 *We illustrate the trace-assertion methodology with a simple example of the parking machine mentioned in the introduction. The machine accepts coins and prints parking permits when a button is pressed. To keep the example simple, we omit other common features of such machines.*

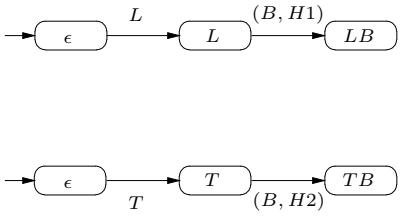

Fig. 1. Basic scenarios.

There are two basic scenarios as shown in Fig. 1, where a transition from a state under input X is labelled by X, if there is no output, and by (X, Y), if there is an output Y. In the initial state, no actions have been taken; this is represented by the empty word ϵ. The coins that are accepted are the Canadian one- and two-dollar coins: a looney, L, and a twoney, T. In the first scenario, a looney is inserted, and then the button B is pushed. The corresponding machine response is a permit for one hour, denoted by H_1. In the second scenario, when a twoney is inserted and the button pushed, a two-hour permit H_2 is issued. These two scenarios are combined into one use case in Fig. 2.

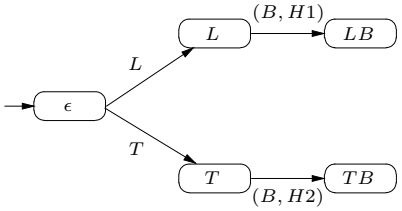

Fig. 2. Use case.

In this example, there is no obvious representation of the state of the machine. Therefore, the input traces constitute a natural choice. Thus we have the five traces ϵ, L, T, LB, TB—these are the canonical traces.

Next, we need to determine what should happen when other inputs are applied. For example, when a passer-by presses the button B, there should be no change of state. If a customer inserts two looneys, this should be equivalent to the insertion of a twoney. If the customer does not have the correct change and inserts a looney followed by a twoney, a refund R_1 of one dollar is made. If more than two dollars are inserted, the overpayment is refunded. Once the permit has been printed, no action is taken if the button is pressed. The complete set of equivalences is shown in Fig. 3.

In a formal specification of this module we list the following items:

Input Alphabet $\{L, T, B\}$.

Output Alphabet $\{H_1, H_2, R_1, R_2\}$.

Syntax $B, L, T : \langle \texttt{parkingmachine} \rangle \to \langle \texttt{parkingmachine} \rangle \times (\langle \texttt{output} \rangle \cup \{\epsilon\})$.

Canonical Traces $\{\epsilon, L, T, LB, TB\}$.

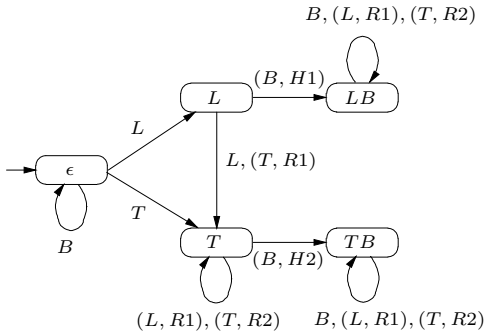

Fig. 3. Complete parking machine.

Trace Equivalence $\mathbf{G} = \{(B, \epsilon), (LL, T), (LT, T), (TL, T), (TT, T), (LBB, LB), (LBL, LB), (LBT, LB), (TBB, TB), (TBL, TB), (TBT, TB)\}$.

Rewriting System $\mathbf{T} = \{Bx \models x, LLx \models Tx, LTx \models Tx, TLx \models Tx, TTx \models Tx, LBBx \models LBx, LBLx \models LBx, LBTx \models LBx, TBBx \models TBx, TBLx \models TBx, TBTx \models TBx\}$

Legality *All traces are legal.*

Outputs $\nu(LB) = H_1$, $\nu(LL) = \epsilon$, $\nu(TB) = H_2$, $\nu(LT) = R_1$, *etc., as in Fig. 3.*

5. Finite versus Infinite Specifications

Although practical software modules are necessarily finite, we consider both finite and infinite specifications for two reasons. First, our theory works equally well in both cases, and, second, an infinite specification is often simpler than a finite one and may be used as an intermediate step in finding a finite one. We illustrate these ideas with an infinite, and then a finite, stack module.

5.1. An Infinite Stack

This stack module is based on that of [1]. The stack is initially empty. We can push any integer z onto the stack using operation $\texttt{push}(z)$, 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 are illegal. The `depth` operation d returns the number of integers stored on the stack, when d follows a legal trace.

We now state the complete set of trace assertions for the stack. The input alphabet is the infinite set $\Sigma = \{d, p, t\} \cup Z$, the mark alphabet is $\Xi = \{\texttt{legal}, \texttt{illegal}\}$, and the output alphabet is $\Omega = Z$. The syntax is

$$p, z : \langle \texttt{stack} \rangle \to \langle \texttt{stack} \rangle, \qquad \forall z \in Z,$$
$$d, t : \langle \texttt{stack} \rangle \to \langle \texttt{stack} \rangle \times (Z \cup \{\epsilon\}).$$

A natural set of canonical words to represent the contents of the stack is Z^*. All these words are legal. Note that a canonical word $w \in Z^*$ followed by any $z \in Z$

results in the canonical word wz. This corresponds to a stack storing the integers appearing in wz with z acting as the top of the stack.

An error situation arises for the two words p and t. Either one could be chosen as canonical, if we do not want to distinguish these words as two different types of errors; we choose p.

The complete set $\mathbf{X} = Z^* \cup \{p\}$ of canonical words is prefix-continuous, in fact, prefix-closed.

As explained in [7], the equivalences are obtained as follows. Extend a canonical word w by a letter $a \in \Sigma$. If wa is canonical, no equivalence is generated. Otherwise, find the canonical representative of wa. The equivalences are:

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

The equivalence **E1** expresses that the extension of the empty word by t is illegal, since we are asking for the top of an empty stack. Equivalence **E2** asserts that the stack is not changed by the `depth` operation. In **E3** we state that any extension pa of p is illegal; in this example we assume that we have no way of recovering from an error. Equivalence **E4** declares that the top operation applied to a nonempty stack does not change the stack. Finally, **E5** asserts that a `pop` following a `push` cancels the `push`.

We now verify that we have considered all the possibilities. The extensions of the empty word by p and z are canonical; hence no equivalence is needed. The extensions of the empty word by t and d are considered in **E1** and **E2**, respectively. Now consider any canonical $w \in Z^*$. Its extension by z is canonical, whereas the extensions by d, t and p are accounted for in **E2**, **E4** and **E5**, respectively. The extension of p by any letter is handled by **E3**.

Since ϵ is canonical, $\hat{\mathbf{G}} = \mathbf{G} = \{\mathbf{E1}, \ldots, \mathbf{E5}\}$. From these equivalences one immediately obtains the rewriting system $\hat{\mathbf{T}} = \mathbf{T} = \{\mathbf{T1}, \ldots, \mathbf{T5}\}$, where

T1	(t, p),	
T2	(wd, w),	
T3	(pa, p),	$\forall a \in \Sigma$,
T4	(wzt, wz),	$\forall w \in Z^*, z \in Z$,
T5	(wzp, w),	$\forall w \in Z^*, z \in Z$.

Among the words in $\mathbf{X}$, the words in Z^* are all legal, whereas p is illegal. Thus we set $\mu(w) =$ `legal` if $w \in Z^*$, and $\mu(p) =$ `illegal`.

Only inputs d and t produce outputs. Input d following any legal canonical word w results in the output value $|w|$. Input t applied after any word of the form wz, with $w \in Z^*$ and $z \in Z$, results in the output value z. Formally, the output function ν is defined by

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

At this point we have sufficient information to construct the stack automaton.

Definition 1 *The* stack automaton *is* $M = (\Sigma, Q, \delta, q_\epsilon, \Xi, \mu, \Omega, \nu)$, *where* $\Sigma = \{d, p, t\} \cup Z$, $Q = Z^* \cup \{p\}$, $q_\epsilon = \epsilon$, $\Xi = \{\texttt{legal}, \texttt{illegal}\}$, $\Omega = Z$, *and* δ, μ, *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	$\delta(q, z) = qz$,	$\forall q \in Z^*, z \in Z$,
C2	$\delta(\epsilon, p) = p$,	
N1	$\delta(\epsilon, t) = p$,	
N2	$\delta(q, d) = q$,	$\forall q \in Z^*$,
N3	$\delta(p, a) = p$,	$\forall a \in \Sigma$,
N4	$\delta(qz, t) = qz$,	$\forall q \in Z^*, z \in Z$,
N5	$\delta(qz, p) = q$,	$\forall q \in Z^*, z \in Z$,
L1	$\mu(q) = \texttt{legal}$,	$\forall q \in Z^*$,
L2	$\mu(p) = \texttt{illegal}$,	
O1	$\nu(q, d) = \lvert q \rvert$,	$\forall q \in Z^*$,
O2	$\nu(qz, t) = z$,	$\forall q \in Z^*, z \in Z$.

The transitions of the stack automaton are of two types, labelled **Ci** and **Ni**. The first type corresponds to an extension of a canonical word w by a letter a, where wa is again canonical. Transitions of the type **Ni** are obtained directly from equivalences **Ei**. By a legal state we mean a state q such that $\mu(q) = \texttt{legal}$.

The stack automaton is illustrated in Fig. 4. As before, a transition from state q under input a is labelled by a, if there is no output, and by (a, b), if there is an output b. The marking function is not shown in the figure. 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.

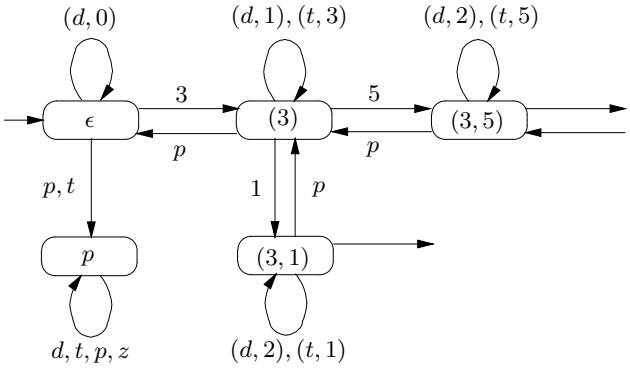

Fig. 4. Stack automaton.

The stack automaton is reduced, as we now showh. Since $\mu(p) \neq \mu(q)$ for all states $q \in Z^*$, state p is not equivalent to any other state. Among the legal 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, as $\delta(q, p^j) = p$ and $\delta(q', p^j) = \epsilon$. 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 $p^{n-i}t$.

Since the stack automaton is reduced, there is no finite automaton by which the stack module can be specified. The automaton is infinite in two ways: 1) it has an infinite number of states, as the depth of the stack is unbounded, and 2) it can store arbitrarily large integers, that is, its input and output alphabets are infinite. Nevertheless, we were able to find a finite representation of this automaton by specifying a finite number of patterns of equivalences and rewriting rules.

5.2. Finite Stacks

In practice, the size of the stack is limited by some maximum capacity $n > 0$, and the nature of the items to be stored. Let $B = \{z \mid 0 \le z \le b\}$, and assume that only integers in B can be pushed onto the stack. It is still possible for the user to attempt pushing an integer $z \notin B$, but the stack module will then enter the illegal state. It is also illegal to push an integer if the stack is full, that is, has depth n.

We leave the syntax of the finite stack the same as that of the infinite one. In that case illegal inputs must be detected by the module, as reflected in the equivalences below.

Let $B_n = \bigcup_{i=0}^{n} B^i$; as the set of canonical traces we choose $\mathbf{X} = B_n \cup \{p\}$. The equivalences are:

E1	$wz \equiv p,$	if $(w \in B^n, z \in Z)$ or $(w \in B_n, z \in Z \setminus B)$,
E2	$t \equiv p,$	
E3	$wd \equiv w,$	$\forall w \in B_n,$
E4	$pa \equiv p,$	$\forall a \in \Sigma,$
E5	$wzt \equiv wz,$	$\forall w \in B_{n-1}, z \in B,$
E6	$wzp \equiv w,$	$\forall w \in B_{n-1}, z \in B.$

The rewriting rules follow directly from the basic equivalences. The formal definition of the automaton is as follows: $M = (\Sigma, Q, \delta, q_\epsilon, \Xi, \mu, \Omega, \nu)$, where $\Sigma = \{d, p, t\} \cup Z$, $Q = B_n \cup \{p\}$, $q_\epsilon = \epsilon$, $\Xi = \{\texttt{legal}, \texttt{illegal}\}$, $\Omega = B \cup \{0, \ldots, n\}$, and δ, μ, and ν are defined below.

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	$\delta(p, a) = p,$	$\forall a \in \Sigma,$
N5	$\delta(qz, t) = qz,$	$\forall q \in B_{n-1}, z \in B,$
N6	$\delta(qz, p) = q,$	$\forall q \in B_{n-1}, z \in B,$
L1	$\mu(q) = \texttt{legal},$	$\forall q \in B_n,$
L2	$\mu(p) = \texttt{illegal},$	
O1	$\nu(q, d) = \lvert q \rvert,$	$\forall q \in B_n,$
O2	$\nu(qz, t) = z,$	$\forall q \in B_{n-1}, z \in B.$

The semiautomaton of the finite stack is shown in Fig. 5, where $n = 2$ and $b = 1$. Transitions under integer inputs that are not in B are not shown; they all lead to p.

Alternatively, we could expect the user of the module to apply only correct parameters. The input alphabet is now $\Sigma = \{d, p, t\} \cup B$. The syntax is

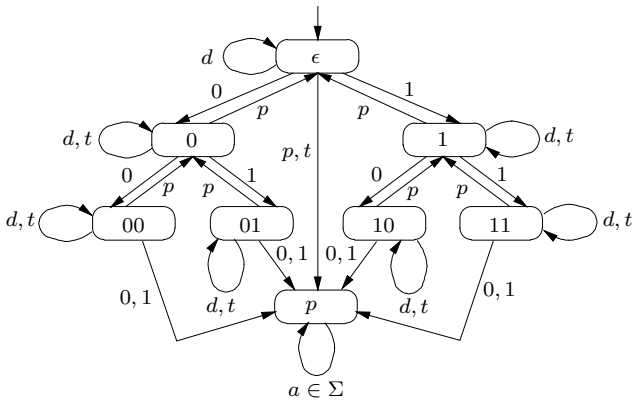

Fig. 5. A finite stack semiautomaton.

$p, z : \langle \texttt{stack} \rangle \to \langle \texttt{stack} \rangle, \qquad \forall z \in B,$
$d : \langle \texttt{stack} \rangle \to \langle \texttt{stack} \rangle \times (\{0, \ldots, n\} \cup \{\epsilon\})$
$t : \langle \texttt{stack} \rangle \to \langle \texttt{stack} \rangle \times (B \cup \{\epsilon\}).$

The equivalences now become

E1	$wz \equiv p,$	$\forall w \in B^n, z \in B,$
E2	$t \equiv p,$	
E3	$wd \equiv w,$	$\forall w \in B_n,$
E4	$pa \equiv p,$	$\forall a \in \Sigma,$
E5	$wzt \equiv wz,$	$\forall w \in B_{n-1}, z \in B,$
E6	$wzp \equiv w,$	$\forall w \in B_{n-1}, z \in B.$

The illustration in Fig. 5 still applies. In both cases, the automaton is reduced.

6. Error-Handling

It may be desirable to classify the errors that may occur; this can be done using the marking function. For example, in a stack, states in which no errors occurred are marked `legal`. An attempt to pop an empty stack or to ask for its top leads to a state marked `underflow`. If the stack is full, a push attempt leads to a state marked `overflow`. Finally, an attempt to push an integer outside of B onto a legal stack leads to a state marked `input_error`. When an attempt is made to push an integer not in B onto a full stack, we arbitrarily choose the error message `input_error`.

The canonical traces are $\mathbf{X} = B_n \cup \{p, -1, 0^{n+1}\}$, where -1 represents the occurrence of an input error, and 0^{n+1} represents overflow. It is assumed that, once an error of a certain type is detected, the module remains in that error state. In practice, a *reset* input would be required to recover from the error states; we do not include these details.

The equivalences now become

E1	$wz \equiv -1$,	$\forall w \in B_n, z \in Z \setminus B$,
E2	$-1a \equiv -1$,	$\forall a \in \Sigma$,
E3	$wz \equiv 0^{n+1}$,	$\forall w \in B^n, z \in B$,
E4	$0^{n+1}a \equiv 0^{n+1}$,	$\forall a \in \Sigma$,
E5	$t \equiv p$,	
E6	$pa \equiv p$,	$\forall a \in \Sigma$,
E7	$wd \equiv w$,	$\forall w \in B_n$,
E8	$wzt \equiv wz$,	$\forall w \in B_{n-1}, z \in B$,
E9	$wzp \equiv w$,	$\forall w \in B_{n-1}, z \in B$.

The construction of the semiautomaton is now straightforward. We use the mark alphabet $\Xi = \{\texttt{legal}, \texttt{underflow}, \texttt{overflow}, \texttt{input\_error}\}$, and the marking function

$$\mu(q) = \begin{cases} \texttt{legal}, & \text{if } q \in B_n, \\ \texttt{underflow}, & \text{if } q = p, \\ \texttt{overflow}, & \text{if } q = 0^{n+1}, \\ \texttt{input\_error}, & \text{if } q = -1. \end{cases}$$

One verifies that this automaton is reduced.

7. Trace-Assertion Specifications versus Automata

In the case of stacks, the trace-assertion specification leads to a natural representation of states by input words. Other examples of this kind include the counter and the shift-register described in [7]. Below we provide an additional example, the specification of a queue, like that of [1], but in our formalism.

There are also cases when the specification by trace assertions is quite awkward, whereas the automaton model is more natural. This is illustrated by the examples of set, sorting queue, and linked list in Sections 7.2–7.4. We use a different approach in these sections. First, we construct an automaton for the given module, and select a set of canonical traces using a spanning forest of the automaton. We then claim that a trace-assertion specification can always be constructed, and indicate how this can be done. However, in our view, the automaton specification is more natural.

7.1. Queue

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. The syntax is

$r, z : \langle\texttt{queue}\rangle \to \langle\texttt{queue}\rangle, \qquad \forall z \in Z,$
$f : \langle\texttt{queue}\rangle \to \langle\texttt{queue}\rangle \times (Z \cup \{\epsilon\}).$

We choose $Z^* \cup \{r\}$ as the set of canonical traces. Thus, when $w \in Z^*$, the canonical trace corresponds to the contents of the queue. The set $Z^* \cup \{r\}$ is prefix-closed. The equivalences are

E1	$f \equiv r$,	
E2	$ra \equiv r$,	$\forall a \in \Sigma$,
E3	$zwf \equiv zw$,	$\forall w \in Z^*, z \in Z$,
E4	$zwr \equiv w$,	$\forall w \in Z^*, z \in Z$.

From here one constructs the automaton, which is reduced. As in the case of a stack, we can change the specification to a finite one, and introduce more detailed error handling. The construction of the automaton, including these modifications, is quite straightforward, once one has a prefix-continuous set of canonical words and the equivalences.

7.2. Set

This example is derived from the "intset" example of [11], discussed also in [26]. 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 $z \in S$. The operation `delete`(z), denoted by $\bar{z}$, removes z from S, and does nothing if $z \notin S$. The operation `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. 6.

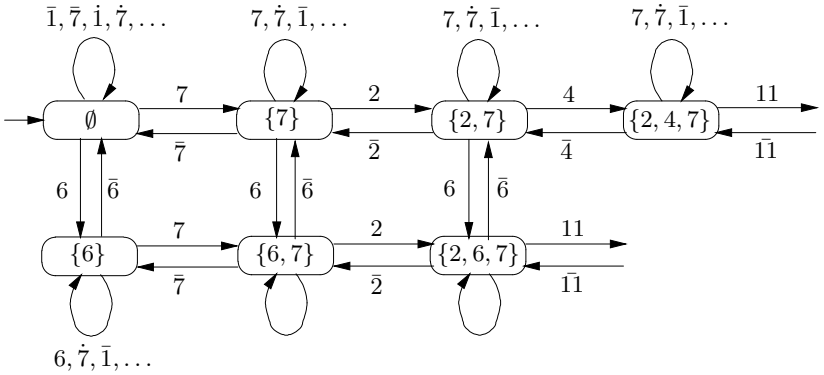

Fig. 6. Set semiautomaton.

Definition 2 *The* set automaton *is* $M = (\Sigma, Q, \delta, q_\epsilon, \Xi, \mu, \Omega, \nu)$*, where* $\Sigma = Z \cup \bar{Z} \cup \dot{Z}$*,* Q *is the set of all finite subsets of* Z*,* $q_\epsilon = \emptyset$*,* $\Xi = \{\texttt{legal}\}$*,* $\Omega = \{\texttt{true}, \texttt{false}\}$*, and* δ*,* μ *and* ν *are defined below.*

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	$\delta(q,\dot{z}) = q,$	$\forall q \in Q, z \in Z,$
L	$\mu(q) = \texttt{legal},$	$\forall q \in Q,$
O	$\nu(q,\dot{z}) = \begin{cases} \texttt{true}, & if q \in Q, z \in q, \\ \texttt{false}, & if q \in Q, z \notin q. \end{cases}$	

In constructing a trace-assertion specification, our first problem is the selection of canonical traces. Here the choice is not as obvious as it was in the case of a stack or a queue, and there are two natural options: To represent a given set by a trace, we can arrange its elements in either increasing or decreasing order. We choose the decreasing order here.

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 $set(w) = \{z_1, \ldots, z_n\}$.

The set of canonical words is $setsort(Q)$; this set is prefix-closed. The equivalences are

E1	$wz \equiv w,$	$\forall w \in setsort(Q), z \in Z, z \in set(w),$
E2	$w\bar{z} \equiv setsort(set(w) \setminus \{z\}),$	$\forall w \in setsort(Q), z \in Z,$
E3	$w\dot{z} \equiv w,$	$\forall w \in setsort(Q), z \in Z.$

From here on, one can easily construct the automaton corresponding to this choice of canonical traces. One verifies that the two automata are isomorphic and reduced. The restriction of the mapping *set* to $setsort(Q)$ constitutes an isomorphism between the two automata.

7.3. Sorting Queue

This example is derived from [1]. A *squeue* (sorting queue) is either empty or is a multiset (bag) of integers (duplicates are permitted). If the squeue is nonempty, `remove`, denoted by r, removes one occurrence of the largest integer in the squeue. Otherwise, `remove` is illegal. If the squeue is nonempty, `max`, denoted by m, returns the largest integer in the squeue without changing the squeue. For each integer $z \in Z$, `insert`(z), denoted by z, inserts z in the squeue.

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 coefficient 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 squeue semiautomaton is illustrated in Fig. 7.

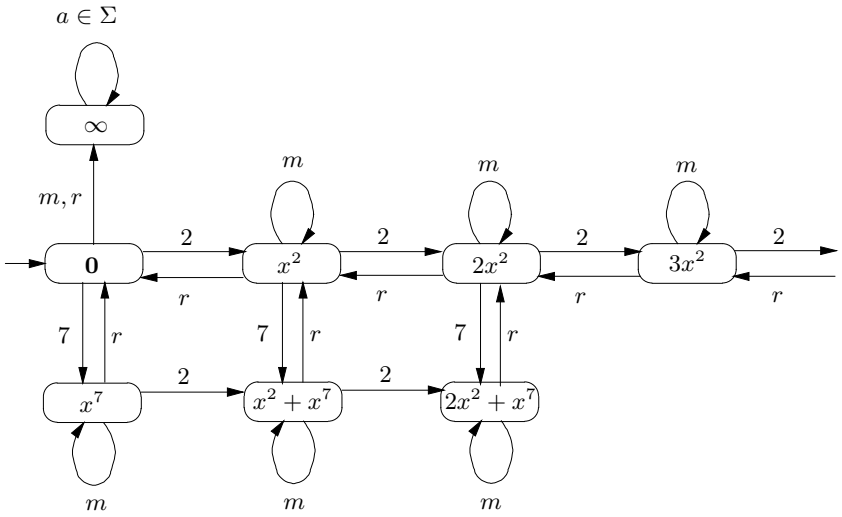

Fig. 7. Squeue semiautomaton.

Definition 3 *The* squeue automaton *is* $M = (\Sigma, Q, \delta, q_\epsilon, \Xi, \mu, \Omega, \nu)$, *where* $\Sigma = \{m, r\} \cup Z$, $Q = Q' \cup \{\infty\}$, Q' *is the set of all finite multisets over* Z, $q_\epsilon = \mathbf{0}$, $\Xi = \{\texttt{legal}, \texttt{illegal}\}$, $\Omega = Z$, *and* δ, μ *and* ν *are defined below.*

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	$\delta(x^z + \sigma, r) = x^z + \sigma - x^{\max(x^z + \sigma)}$,	$\forall \sigma \in Q', z \in Z$,
L1	$\mu(\sigma) = \texttt{legal}$,	$\forall \sigma \in Q'$,
L1	$\mu(\infty) = \texttt{illegal}$,	
O	$\nu(x^z + \sigma, m) = \max(x^z + \sigma)$,	$\forall \sigma \in Q', z \in Z$.

As in the case of the set module, there are two natural choices of canonical traces. The elements can be listed in either non-increasing or non-decreasing order; we choose the former. Define function $sort : Z^* \to Z^*$ as: $sort(\epsilon) = \epsilon$, 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 $sort(Z^*) = \{sort(z) \mid z \in Z^*\}$.

Let r be the canonical word for ∞. The set $sort(Z^*) \cup \{r\}$ is the set of canonical words, and it is prefix-closed.

The equivalences are:

E1	$wz \equiv sort(wz)$,	$\forall w \in sort(Z^*), z \in Z, wz \neq sort(wz)$
E2	$m \equiv r$,	
E3	$ra \equiv r$,	$\forall a \in \Sigma$,
E4	$zwm \equiv zw$,	$\forall w \in Z^*, z \in Z, zw \in sort(Z^*)$,
E5	$zwr \equiv w$,	$\forall w \in Z^*, z \in Z, zw \in sort(Z^*)$.

As before, one can easily construct the automaton corresponding to this choice of canonical traces. Again the two automata are isomorphic and reduced.

7.4. Linked List

This example is similar to the Table/List of [1]. A linked list, which we call *llist*, consists of a list of integers and a *current pointer* into the list. The current pointer ($\downarrow$) can point to any position in the list, including the position just to the right of the list. In particular, when the list is empty, the current pointer is just to the right of the empty list; we denote the empty llist by $\downarrow$. If the llist is not empty, the element just to the right of the pointer is the *current element.* For example, the notation $z_4z_1z_3 \downarrow z_1z_2$ means that the list contains z_4, z_1, z_3, z_1, z_2 in the order shown, and the current pointer points to the fourth element in the list, that is, the second occurrence of z_1.

The llist is initially empty. The `insert`(z) operation, denoted by z, inserts z to the left of the current pointer, and moves the pointer one position to the left, so that z becomes the current element. In the example above, inserting z results in the new llist is $z_4z_1z_3 \downarrow zz_1z_2$. Inserting z into the empty llist, yields $\downarrow z$. Operation `left`, denoted l, moves the current pointer to the left, provided there is at least one element to the left of the pointer. For example, applying l to $z_4z_1z_3 \downarrow z_1z_2$ we get $z_4z_1 \downarrow z_3z_1z_2$. Applying l to $\downarrow$ or $\downarrow z_4z_1z_3z_1z_2$ is illegal. Operation `right`, denoted r, moves the current pointer to the right, provided there is at least one element to the right of the pointer. For example, applying r to $z_4z_1z_3 \downarrow z_1z_2$ we get $z_4z_1z_3z_1 \downarrow z_2$. Applying r to $\downarrow$ or $z_4z_1z_3z_1z_2 \downarrow$ is illegal. Operation `delete`, denoted d, removes the element to the right of the pointer, if such an element exists, and the current pointer does not move. For example, applying d to $z_4z_1z_3 \downarrow z_1z_2$ we get $z_4z_1z_3 \downarrow z_2$, and applying d to $\downarrow z$ yields $\downarrow$. Applying d to $\downarrow$ or $z_4z_1z_3z_1z_2 \downarrow$ is illegal. The llist also has operation `current`, denoted by c, which returns the value of the current element, if there is one, and is illegal, otherwise.

To represent a state of a llist automaton, we use a word $(u \downarrow v)$, where u and v are words in Z^*.

Definition 4 *The* llist automaton *is defined as* $M = (\Sigma, Q, \delta, q_\epsilon, \Xi, \mu, \Omega, \nu)$*, where* $\Sigma = \{c, d, l, r\} \cup Z$*,* $Q = (Z^* \downarrow Z^*) \cup \{\infty\}$*,* $q_\epsilon = \downarrow$*,* $\Xi = \{\texttt{legal}, \texttt{illegal}\}$*,* $\Omega = Z$*, and* δ*,* μ *and* ν *are as follows:*

C1	$\delta(\downarrow v, z) = \downarrow zv$,	$\forall v \in Z^*, z \in Z$,
C2	$\delta(u \downarrow zv, r) = uz \downarrow v$,	$\forall u, v \in Z^*, z \in Z$,
C3	$\delta(\downarrow, c) = \infty$,	
N1	$\delta(u \downarrow v, z) = u \downarrow zv$,	$\forall u \in Z^+, v \in Z^*, z \in Z$,
N2	$\delta(uz \downarrow, c) = \infty$,	$\forall u \in Z^*, z \in Z$,
N3	$\delta(u \downarrow, d) = \infty$,	$\forall u \in Z^*$,
N4	$\delta(\downarrow v, l) = \infty$,	$\forall v \in Z^*$,
N5	$\delta(u \downarrow, r) = \infty$,	$\forall u \in Z^*$,
N6	$\delta(\infty, a) = \infty$,	$\forall a \in \Sigma$,
N7	$\delta(u \downarrow zv, c) = u \downarrow zv$,	$\forall u, v \in Z^*, z \in Z$,
N8	$\delta(u \downarrow zv, d) = u \downarrow v$,	$\forall u, v \in Z^*, z \in Z$,
N9	$\delta(uz \downarrow v, l) = u \downarrow zv$,	$\forall u, v \in Z^*, z \in Z$,
L1	$\mu(u \downarrow v) = \texttt{legal}$,	$\forall u, v \in Z^*$,
L2	$\mu(\infty) = \texttt{illegal}$,	
O	$\nu(u \downarrow zv, c) = z$,	$\forall u, v \in Z^*, z \in Z$.

Because the states are not represented by words in Σ^*, it is not obvious how to choose canonical traces. As we now show, a trace-assertion representation of this is rather complicated. The idea underlying our choice of canonical traces is to represent a state by a sequence of insertions followed by a sequence of right moves, to position the pointer correctly.

For $w \in \Sigma^*$, let w^ρ be the reversal of w. For the canonical trace leading to state $u \downarrow v$ 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 $0 \leq k \leq n$. Observe that, when $w = z_1 \dots z_n$ is applied, the resulting state is $\downarrow z_n \dots z_1$. If r is now applied n times, the result is $z_n \dots z_1 \downarrow$. In any such state, operations c, d, and r are illegal, while l results in $z_n \dots z_2 \downarrow z_1$, and z yields $z_n \dots z_1 \downarrow z$. In case $k < n$, the final state is $z_n \dots z_{n-k+1} \downarrow z_{n-k} \dots z_1$. Operations c, d, r and z are legal, and l is legal provided $k > 0$. We are now ready to give state the equivalences.

We introduce the following notation: if $i \leq j$, then $w|_i^j = z_i \dots z_j$. The equivalences are:

E1	$ur^k z \equiv u|_1^{n-k-1} z u|_{n-k}^n r^k$,	$\forall u \in Z^*, z \in Z, 0 < k \leq n = \|u\|$,
E2	$uzr^{\|u\|+1} c \equiv c$,	$\forall u \in Z^*, z \in Z$,
E3	$ur^{\|u\|} d \equiv c$,	$\forall u \in Z^*$,
E4	$ul \equiv c$,	$\forall u \in Z^*$,
E5	$ur^{\|u\|} r \equiv c$,	$\forall u \in Z^*$,
E6	$ca \equiv c$,	$\forall a \in \Sigma$,
E7	$ur^k c \equiv ur^k$,	$\forall u \in Z^*, k < \|u\|$,
E8	$ur^k d \equiv u|_1^{n-k-1} u|_{n-k+1}^n r^k$,	$\forall u \in Z^+, k < n = \|u\|$,
E9	$ur^k l \equiv ur^{k-1}$,	$\forall u \in Z^+, 0 < k < \|u\|$,

Each transition **Ni** corresponds to equation **Ei**. Transitions **C1** and **N1** could have been combined into

$$\delta(u \downarrow v, z) = u \downarrow zv, \quad \forall u \in Z^*, v \in Z^*, z \in Z.$$

We chose to separate them in order to simplify the derivation of equivalences.

8. Conclusions

In [7] we provided a theoretical framework, based on semiautomata, for the trace-assertion specification method. We showed there that finding trace equivalence amounts to constructing a generating set for state equivalence, and we presented a simple algorithm for finding this set. Directly from this generating set, we derived a rewriting system which permits the transformation of any trace to its canonical form. We proved that this rewriting system has no infinite derivations if and only if the canonical set is prefix-continuous.

In the present paper, we have extended the framework to modules with outputs, and have provided a formalism for error handling. Mealy outputs are used for module outputs, whereas Moore outputs permit the classification of errors. We have presented a unified trace-assertion methodology, using our theory as a guide. Our step-by-step description leads to a systematic construction of a trace-assertion specification, and guarantees a well-behaved rewriting system. We have illustrated our approach, and some specific issues, using several nontrivial examples. Our examples constitute a comparison of the two specification methods: trace assertions and automata. While these two specification methods are equivalent, there are cases where one approach is more natural than the other. We conclude that formal specification by automata, as opposed to informal state machines, is not only possible, but practical, and may be preferable to trace assertions.

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

References

1. W. Bartussek and D. L. Parnas, *Using Assertions About Traces to Write Abstract Specifications for Software Modules,* TR77-012 (Univ. of NC, Chapel Hill, 1977).
2. W. Bartussek and D. L. Parnas, Using Assertions About Traces to Write Abstract Specifications for Software Modules, in *Inform. Syst. Methodology,* LNCS **65** (Springer, 1978), 211–236.
3. W. Bartussek and D. L. Parnas, Using Assertions About Traces to Write Abstract Specifications for Software Modules, in *Software Fundamentals (Collected Works by D. L. Parnas),* eds. D. M. Hoffman and D. M. Weiss (Addison-Wesley, 2001), 9–28.
4. R. V. Book and F. Otto, *String-Rewriting Systems* (Springer, 1993).
5. J. A. Brzozowski, Representation of a Class of Nondeterministic Semiautomata by Canonical Words, *Theoretical Computer Science* **356** (2006), 46-57.
6. J. A. Brzozowski and H. Jürgensen, *Theory of Deterministic Trace-Assertion Specifications.* Tech. Rept. CS-2004-30 (School of Comp. Sci., Univ. of Waterloo, 2004). `http://www.cs.uwaterloo.ca/cs-archive/CS-2004/CS-2004.shtml`
7. J. A. Brzozowski and H. Jürgensen, Representation of Semiautomata by Canonical Words and Equivalences, *Int. J. Found. of Computer Science* **16**(5) (2005), 831–850.
8. J. R. Büchi, Regular Canonical Systems, in *Archiv für Math. Logik und Grundlagenforschung* **6** (1964), 91–111.
9. J. R. Büchi, *Finite Automata, Their Algebras and Grammars,* ed. D. Siefkes (Springer 1988).

10. M. Fowler and K. Scott, *UML Distilled – A Brief Guide to the Standard Object Modeling Language,* 2nd edition (Addison-Wesley, 2000).

11. J. V. Guttag, E. Horowitz and R. Musser, The Design of Data Type Specifications, in *Current Trends in Programming Methodology* **IV,** ed. R. T. Yeh (Prentice-Hall, 1978), 60–79.

12. D. Hoffman, The Trace Specification of Communications Protocols, *IEEE Trans. Computers* **C34**(12) (1985), 1102–1113.

13. D. Hoffman and R. Snodgrass, Trace Specifications: Methodology and Models. *IEEE Trans. Software Engineering* **14**(9) (1988), 1243–1252.

14. M. Iglewski, M. Kubica and J. Madey, *Trace Specifications of Nondeterministic Multi-Object Modules,* Tech. Rept. TR 95-05 (205) (Institute of Informatics, Warsaw University, Warsaw, Poland, 1995).

15. M. Iglewski, M. Kubica, J. Madey, J. Mincer-Daszkiewicz and K. Stencel, *TAM'97: The Trace Assertion Method of Module Interface Specification,* Reference Manual (1997), `http://w3.uqah.uquebec.ca/iglewski/public_html/TAM/`

16. R. Janicki and E. Sekerinski, Foundations of the Trace Assertion Method of Module Interface Specifications, *IEEE Trans. Software Engineering* **27**(7) (2001), 577–598.

17. Z. Kohavi, *Switching and Finite Automata Theory* (McGraw-Hill, 1978).

18. N. Kuhn and K. Madlener, A Method for Enumerating Cosets of a Group Presented by a Canonical System, *Proc. ACM-SIGSAM Int. Symp. on Symbolic and Algebraic Computation* (1989), 338–350.

19. J. Madey, From Requirement Analysis to Code Verification—a Functional Approach (in Polish), *Proc. 8th Conference on Real-Time Systems,* Krynica, Sept. 2001, eds. T. Szmuc and R. Klimek (Chair of Automatics, AGH University of Science and Technology, Cracow, Poland, 2002), 35–74.

20. J. McLean, A Formal Method for the Abstract Specification of Software. *J. ACM* **31**(3) (1984), 600–627.

21. J. McLean, Proving Noninterference and Functional Correctness Using Traces, *Journal of Computer Security* **1**(1) (1992), 37–57.

22. J. McLean, A General Theory of Composition for Trace Sets closed under Selective Interleaving Functions, Traces, *Proc. IEEE Symp. on Research in Security and Privacy, Oakland, CA* (1994).

23. D. L. Parnas and Y. Wang, *The Trace Assertion Method of Module Interface Specification,* Tech. Rept. 89–261 (Queen's University, C&IS, Telecom. Res. Inst. of Ontario, Kingston, ON, Canada 1989).

24. W. Snyder, Efficient Ground Completion: An $O(n \log n)$ Algorithm for Generating Reduced Sets of Ground Rewrite Rules Equivalent to a Set of Ground Equations E, in *Proc. RTA-89, Rewriting Techniques and Applications,* ed. N. Dershowitz, LNCS **355** (1989), 419–433.

25. P. H. Starke, *Abstract Automata* (North-Holland, Amsterdam 1972).

26. Y. Wang, *Formal and Abstract Software Module Specifications — A Survey,* Tech. Rept. 91–307 (Comp. and Inf. Sci., Queen's University, Kingston, ON, 1991).

27. Y. Wang and D. L. Parnas, Simulating the Behavior of Software Modules by Trace Rewriting, *IEEE Trans. on Software Engineering* **20**(10) (1994), 750–759.

28. K. Weidenhaupt, K. Pohl, M. Jarke and P. Haumer, Scenarios in System Development: Current Practice, *IEEE Software* **15**(2) (1998), 34–45.