Verification of Trace-Assertion Specifications of 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. In 1977 Bartussek and Parnas proposed a method for abstract specifications of software modules. The method is based on assertions about traces, and uses the concepts of trace legality and trace equivalence. We identify a flaw in the definition of legality and equivalence, and correct it. We then examine the modified method from the point of view of automaton and language theory, using two examples: first, a unary counter, which is a very special case of an unbounded stack, and then the stack itself. In both examples, we prove the correctness of the trace-assertion specification with respect to the automaton specification. These results demonstrate that the original method of Bartussek and Parnas (with some corrections) deserves to be revisited.

1 Introduction

The trace-assertion method for specifying software modules was introduced in 1977 by Bartussek and Parnas [1]; this paper was reprinted in 2001 [3]. A slightly modified version appeared in 1978 [2]. The method has undergone several changes since the original paper: see the papers of Parnas and Wang [7, 9, 10] and the more recent work in [6] for more details.

In the present paper, our interest lies mainly in the original work of Bartussek and Parnas, not its later versions. The model described in [1, 2] is somewhat unusual from a theoretical point of view, and presents interesting challenges. There is no proof of correctness of this model, and we show that no proof is possible. By applying formal methods based on automaton and language theory, we uncover a flaw in the original definition. We correct the flaw in a way which, we believe, reflects the authors' intentions. We provide a formal proof of correctness for the modified trace-assertion specification with respect to the automaton specification for two examples: a unary counter and its generalization, the unbounded stack of [1]. These results demonstrate that the original method of Bartussek and Parnas (with some corrections) deserves to be revisited.

The remainder of the paper is structured as follows. Section 2 gives some terminology and notation. Section 3 introduces the Bartussek-Parnas model, using a unary counter as an example, and points out a flaw in the definition. A corrected version is then given, as well as a corrected version of a stack. In Section 4, we specify the unary counter by an automaton. In Section 5 we provide a formal proof of correctness of the modified trace-assertion specification with respect to the automaton. In Section 6 we specify the stack by an automaton. The trace-assertion specification of the stack is then verified in Section 7. Concluding remarks are made in Section 8.

2 Terminology and Notation

We denote by Z and P the sets of integers and nonnegative integers, respectively. If Σ is an alphabet, then Σ^+ and Σ^* denote the free semigroup and the free monoid, respectively, generated by Σ. The empty word is ϵ. For $w \in \Sigma^*$ and $a \in \Sigma$, $|w|$ denotes the length of w, and $|w|_a$, the number of times a appears in w.

By a *deterministic automaton*, or simply *automaton*, we mean a tuple $A = (\Sigma, Q, \delta, q_0, F)$, where Σ is a nonempty input alphabet, Q is a nonempty set of states, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. 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 word $w \in \Sigma^*$ is accepted by A if and only if $\delta(q_0, w) \in F$. The language accepted by A is $L(A) = \{w \mid \delta(q_0, w) \in F\}$.

By a *Mealy automaton* M we mean a deterministic automaton with an output alphabet and an output function. More precisely,

$$M = (\Sigma, Q, \delta, q_0, F, \Omega, \nu),$$

where $(\Sigma, Q, \delta, q_0, F)$ is a deterministic automaton, Ω is the output alphabet, and $\nu : Q \times \Sigma \to \Omega$ is a partial function, the output function.

For additional material on automata, see, for example, [5, 8, 11].

3 The Bartussek-Parnas Model

The first example given in [1] is that of a pushdown stack for integer values. To introduce the basic model of Bartussek and Parnas, we begin with an even simpler example: we specialize the stack of [1] to a unary counter, and use the counter to illustrate several issues. We then give our modified specification, which is proved to be correct in the sequel. We also give our specification of the stack, and prove it correct later.

3.1 The Unary Counter

Informally, a unary counter is described as follows. It is a pushdown stack, which is initially empty. Only two operations are possible: PUSH and POP. At any time it is possible to PUSH the integer 1 on top of the stack. If the stack contains n 1s, where $n > 0$, it is possible to POP the stack; this operation removes the top 1 from the stack, leaving $(n-1)$ 1s. The count is represented by the number of 1s on the stack.

We now introduce the approach of Bartussek and Parnas, using their notation. The specification is based on *traces*, which are sequences of operations. It has four parts: *syntax, legality, equivalence,* and *values*. For the counter, the *values* part is irrelevant, since our counter returns no values.

In this example, a *trace* is a word over the alphabet $\{\mathrm{PUSH}, \mathrm{POP}\}$. As in [1], to indicate concatenation of traces S and T, the "dot" notation is used, that is, S.T denotes trace S followed by trace T; the empty trace is denoted by $\emptyset$.

Syntax: This part of the specification defines the nature of the operations of the module being specified. In our case, both operations map type $\langle$counter$\rangle$ into the same type.

$$\mathrm{PUSH}, \mathrm{POP} : \langle \text{counter} \rangle \to \langle \text{counter} \rangle.$$

Legality: If S is a trace, we denote by $\lambda(\mathrm{S})$ the assertion that is true if and only if S is legal. Bartussek and Parnas [1] assume the following *general* properties of legal traces:

The empty trace is legal, that is, $\lambda(\emptyset) = \text{true}$.
The prefix of any legal trace is a legal trace, that is, $\lambda(\mathrm{S.T}) \Rightarrow \lambda(\mathrm{S})$.
If a trace cannot be proved legal by the assertions, it is illegal.

Adapting the stack to the counter, we obtain the following *particular* assertion about legality:

$$\lambda(\mathrm{S}) \Rightarrow \lambda(\mathrm{S.PUSH}).$$

The assertions about legality define only a subset of the set of legal traces. Other legal traces may be implied by the equivalence relation defined next.

Equivalence: Equivalence of traces is denoted by $\equiv$, which is an equivalence relation on the set of all traces with the following *general* property:

S and S$'$ are equivalent if and only if, for all traces T, S.T and S$'$.T are either both legal or both illegal. In symbols,

$$\mathrm{S} \equiv \mathrm{S}' \Leftrightarrow \lambda(\mathrm{S.T}) = \lambda(\mathrm{S}'.\mathrm{T}).$$

In particular, when T is the empty trace, we have

$$\mathrm{S} \equiv \mathrm{S}' \Rightarrow \lambda(\mathrm{S}) = \lambda(\mathrm{S}').$$

For the counter, we also have the following *particular* property:

$$\mathrm{S.PUSH.POP} \equiv \mathrm{S}.$$

Values: Bartussek and Parnas use two types of "access programs" (which we call operations): O-functions and V-functions. V-functions return values, whereas O-functions only change the internal state. If S is a trace and X is a V-function (output-returning operation), then V(S.X) is the value returned when X is applied after S. For the unary counter, there are no values.

The definition in [2] is the same as that in [1]. This definition presents the following problems. The general assertion about equivalence implies that equivalence affects legality and vice versa. Moreover, equivalence of two traces cannot be determined in finitely many steps.

One could also interpret the general assertion about equivalence as simply the implication

$$\mathrm{S} \equiv \mathrm{S}' \Rightarrow \lambda(\mathrm{S.T}) = \lambda(\mathrm{S'.T}).$$

This also fails. For example, PUSH.PUSH.POP.POP is then not equivalent to the empty trace, as one would like it to be.

In the 1989 technical report by Parnas and Wang [7], the definitions are changed. Here, the equivalence of two traces S and S' implies that both S and S' are legal, instead of simply having the same legality. This is unnecessarily restrictive. Furthermore, equivalence implies $\mathrm{V(S)} = \mathrm{V(S')}$. This is incorrect, since two equivalent traces may end in different operations. For example, in the push-down stack described below, traces PUSH(a).TOP and PUSH(a).DEPTH are equivalent, but produce different outputs. The report [7] also introduces canonical traces. One member from each equivalence class is selected and declared canonical, provided that every legal trace is equivalent to exactly one canonical trace. In the case of the counter and the stack, we show that canonical traces are naturally derived from the equivalence relation.

The trace-assertion method of [1, 2] is also briefly described in a 1991 survey by Wang [9] with further modifications. In the notation of [1, 2], these are as follows. The general equivalence property is defined as

$$\mathrm{S} \equiv \mathrm{S}' \Rightarrow \lambda(\mathrm{S}) = \lambda(\mathrm{S}').$$

This is indeed a step in the right direction, and we also adopt this assertion. As we shall see later, however, an important assumption about equivalence, the *right-congruence* property:

$$\mathrm{S} \equiv \mathrm{S}' \Rightarrow \mathrm{S.T} \equiv \mathrm{S'.T},$$

is still missing. There is also a problem with the "values" part. Wang states that $\mathrm{S} \equiv \mathrm{S}'$ implies $\lambda(\mathrm{S}) \Rightarrow \mathrm{V(S.T.X)} = \mathrm{V(S'.T.X)}$, for all T, and for all output-returning operations (V-functions) X. Here, although S is legal, S.T or S.T.X may not be legal.

The right-congruence property is explicitly stated in the 1994 paper by Wang and Parnas [10]. This work, as well as the later work in TAM'97 [6], continue to use canonical traces introduced in [7].

Our own solution takes the following form.

Syntax: The PUSH and POP operations are maps of the type

$\mathrm{PUSH}, \mathrm{POP} : \langle\text{counter}\rangle \to \langle\text{counter}\rangle.$

Legality: For all traces S, S', there is the general assertion

L: $\mathrm{S} \equiv \mathrm{S}' \Rightarrow \lambda(\mathrm{S}) = \lambda(\mathrm{S}')$, and two special assertions
L0: $\lambda(\emptyset) = \text{true}$,
L1: $\lambda(\mathrm{S}) \Rightarrow \lambda(\mathrm{S.PUSH})$.

Equivalence: The equivalence relation is the smallest equivalence satisfying, for all traces S, S', the general assertion

E: $\mathrm{S} \equiv \mathrm{S}' \Rightarrow \mathrm{S.T} \equiv \mathrm{S}'.\mathrm{T}$, for all traces T, and two special assertions
E0: $\mathrm{POP} \equiv \mathrm{POP.S}$,
E1: $\mathrm{S} \equiv \mathrm{S.PUSH.POP}$.

Note that the definition of equivalence is now independent of that of legality. Moreover, by E0, all illegal traces are equivalent. We prove in Section 5 that this specification is correct with respect to an automaton specification.

3.2 The Stack

The informal description is as follows. We start with an empty stack. We can push any integer a onto the stack; this operation is represented by PUSH(a). The POP operation, legal only when the stack in nonempty, removes the top integer from the stack. The operation DEPTH returns the number of integers currently stored on the stack, when it is in any legal state. The TOP operation, is permitted only if there is at least one integer stored on the stack; it returns the value of the top integer on the stack. Thus, in the stack, DEPTH and TOP are V-functions, whereas PUSH(a) and POP are O-functions.

As a preview, we also state our solution for the stack. The original stack specification suffers from the same flaw concerning legality and equivalence. A corrected version is given below, where S and S' are arbitrary traces.

Syntax:

$\mathrm{POP}, \mathrm{PUSH(a)} : \langle\text{stack}\rangle \to \langle\text{stack}\rangle$, for all $\mathrm{a} \in Z$,
$\mathrm{DEPTH}, \mathrm{TOP} : \langle\text{stack}\rangle \to \langle\text{integer}\rangle.$

Legality:

L: $\mathrm{S} \equiv \mathrm{S}' \Rightarrow \lambda(\mathrm{S}) = \lambda(\mathrm{S}')$,
L0: $\lambda(\emptyset) = \text{true}$,
L1: $\lambda(\mathrm{S}) \Rightarrow \lambda(\mathrm{S.PUSH(a)})$, for all $\mathrm{a} \in Z$,

Equivalence:

E: $\mathrm{S} \equiv \mathrm{S}' \Rightarrow \mathrm{S.T} \equiv \mathrm{S}'.\mathrm{T}$, for all traces T,
E0: $\mathrm{POP} \equiv \mathrm{POP.S} \equiv \mathrm{TOP}$,
E1: $\mathrm{S} \equiv \mathrm{S.PUSH(a).POP}$, for all $\mathrm{a} \in Z$,
E2: $\mathrm{S} \equiv \mathrm{S.DEPTH}$,
E3: $\mathrm{S.PUSH(a).TOP} \equiv \mathrm{S.PUSH(a)}$, for all $\mathrm{a} \in Z$.

Values:

V: $\lambda(\mathrm{S}) \wedge \lambda(\mathrm{S.X}) \wedge (\mathrm{S} \equiv \mathrm{S}') \Rightarrow \mathrm{V(S.X)} = \mathrm{V(S'.X)}$, for any V-function X,
V0: $\mathrm{V(DEPTH)} = 0$,
V1: $\lambda(\mathrm{S}) \Rightarrow \mathrm{V(S.PUSH(a).DEPTH)} = 1 + \mathrm{V(S.DEPTH)}$, for all $\mathrm{a} \in Z$,
V2: $\lambda(\mathrm{S}) \Rightarrow \mathrm{V(S.PUSH(a).TOP)} = \mathrm{a}$, for all $\mathrm{a} \in Z$.

This specification is verified in Section 7.

4 Specification of the Counter by an Automaton

We first translate our informal description of the counter into a specification by an automaton. For convenience, we simplify the notation by representing PUSH by 1 and POP, by 0.

Definition 1. *A* counter *is an automaton* $A = (\Sigma, Q, \delta, q_0, F)$, *where* $\Sigma = \{0, 1\}$, $Q = P \cup \{\infty\}$, $q_0 = 0$, $F = P$, *and for all* $n > 0$ *and* $a \in \Sigma$*:* $\delta(\infty, a) = \infty$, $\delta(0, 0) = \infty$, $\delta(0, 1) = 1$, $\delta(n, 0) = n - 1$, *and* $\delta(n, 1) = n + 1$.

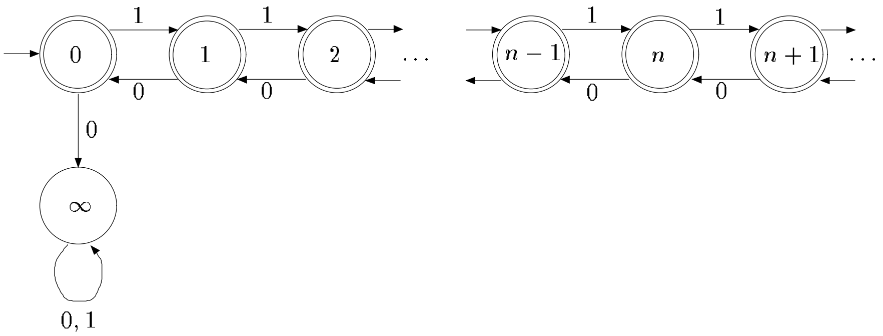

Fig. 1. Automaton of counter

The state graph of A is shown in Fig. 1. The initial state is indicated by an incoming arrow, and accepting states, by double circles. Each transitions between two states is labelled by the input causing this transition. It should be clear that this formal definition corresponds to our informal specification. A stack with n

1s is represented by state n, $n \geq 0$. In particular, the empty stack is represented by state 0. State ∞ represents an illegal state, which is reached when a POP operation (0) is applied to the empty counter.

The specification of the set Λ_A of legal traces of the counter by an automaton is:

$$\Lambda_A = L(A).$$

Note that, if w is in Λ_A, then so is each prefix of w.

The set of all traces is $\Sigma = \{0, 1\}^*$. The set of all legal traces defined by the automaton, $\Lambda_A = L(A)$, is closely related to the Dyck language [4] D of well-formed words over a pair of parentheses, here represented by 0 and 1. In fact, $D = L(A_0)$, where $A_0 = (\Sigma, Q, \delta, 0, \{0\})$ is the counter automaton modified to accept only words leading to the initial state. The language $L(A)$ is the set of all words which are prefixes of words in D. Each of these words can be completed to a well-parenthesized word.

We will require a characterization of the set of words taking automaton A from state 0 to state n with $n \geq 0$. Let $A_n = (\Sigma, Q, \delta, 0, \{n\})$ be the automaton obtained from A by changing the set of accepting states to $\{n\}$. Also, for $n \geq 0$, we denote $L(A_n)$ by Λ_n.

Proposition 1. *The language accepted by A_n is $\Lambda_n = (D1)^n D$. Moreover,*

$$\Lambda_A = \bigcup\nolimits_{n=0}^{\infty} \Lambda_n.$$

Proof. The proof is by induction on n. By the definition of D, the language Λ_0 accepted by A_0 is precisely D, and $D = (D1)^0 D$.

Suppose now that $\Lambda_n = (D1)^n D$, $n > 0$, and consider a word $w \in \Lambda_{n+1}$. Let u be the longest prefix of w that is in Λ_n. The letter following u in w must be 1. Thus let $w = u1v$. In the run of A on w, while v is being read, the state of A must always be some m with $m > n$. As v takes A from state $n+1$ back to itself, we have $v \in D$. Thus $\Lambda_{n+1} \subseteq (D1)^n D1D = (D1)^{n+1} D$. The reverse inclusion is obvious.

Clearly, the language accepted by A is the union of the Λ_n. □

Some important properties of automata are discussed next. The well-known Nerode equivalence $\equiv_X$ of a language X over an alphabet Σ is defined as follows [8]:

$$w \equiv_X w' \text{ if and only if, for all } u \in \Sigma^*,\ wu \in X \Leftrightarrow w'u \in X.$$

The equivalence class of $\equiv_X$ containing w is $[w]_X$.

A (finite or infinite) automaton $A = (\Sigma, Q, \delta, q_0, F)$ is *reduced* if, whenever u and v take the initial state of A to two different states, then u and v are distinguishable (not equivalent) with respect to the Nerode equivalence $\equiv_{L(A)}$, where $L(A)$ is the language accepted by A. In symbols,

$$\delta(q_0, u) \neq \delta(q_0, v) \Rightarrow u \not\equiv_{L(A)} v.$$

Equivalently, A is reduced if

$$u \equiv_{L(A)} v \Rightarrow \delta(q_0, u) = \delta(q_0, v).$$

Given a language X, we obtain its reduced (finite or infinite) automaton $A_X = (\Sigma, Q_X, \delta_X, q_{0,X}, F_X)$ accepting X as follows. The state set Q_X is the set of equivalence classes of $\equiv_X$. The initial state $q_{0,X}$ is the class of ϵ, the set F_X of final states is the set of the classes of words in X, and the transition function is $\delta_X([w]_X, a) = [wa]_X$.

The following is easily verified:

Proposition 2. *The counter automaton A of Fig. 1 is reduced.*

Let $||w|| = |w|_1 - |w|_0$. The next result is well known in language theory [4].

Proposition 3. *A word w is accepted by automaton A if and only if no prefix u of w has more* 0*s than* 1*s, that is,* $L(A) = \{w \mid w = uv, \text{ implies } ||u|| \geq 0\}$.

Note that one could specify the unary counter as follows:

A trace w is legal if and only if $||u|| \geq 0$, for every prefix u of w.

This specification is a type of trace-assertion specification, though, of course, not in the style of Bartussek and Parnas.

5 Verification of the Unary Counter

In this section we prove the correctness of our modified trace-assertion specification of the unary counter. For convenience, we use the notation of the previous section. The specification takes the following form:

Syntax:

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

Legality: For all $w, w' \in \Sigma^*$,

L: $w \equiv w' \Rightarrow \lambda(w) = \lambda(w')$,
L0: $\lambda(\epsilon) = \text{true}$,
L1: $\lambda(w) \Rightarrow \lambda(w1)$, for all $w \in \Sigma^*$.

Equivalence: The equivalence relation is the smallest equivalence satisfying, for all $w, w' \in \Sigma^*$,

E: $w \equiv w' \Rightarrow wu \equiv w'u$, for all $u \in \Sigma^*$,
E0: $0 \equiv 0w$,
E1: $w \equiv w10$.

First we show that every word $w \in \Sigma^*$ has a canonical representative $\hat{w}$ such that $w \equiv \hat{w}$.

Proposition 4. *For every $w \in \Sigma^*$, either $w \equiv 0$, or there is an $i \in P$ such that $w \equiv 1^i$.*

Proof. We repeatedly apply the following $\equiv$-equivalence-preserving transformations to w:

1. If $w = 1^j 0 v$ for some $j > 0$ and $v \in \Sigma^*$, then $w \rightarrow 1^{j-1} v$.
2. If $w \in 0\Sigma^*$, then $w \rightarrow 0$.

Transformation 1 preserves $\equiv$ because we have $1^j 0 = 1^{j-1}(10) \equiv 1^{j-1}$ by E1, and then $1^j 0 v \equiv 1^{j-1} v$ by E. Transformation 2 preserves $\equiv$ by E0. Note that no transformation is applicable if $w = 0$, or $w = 1^j$ for $j \geq 0$. Thus the process stops with one of these words. □

Proposition 5. *Let $\equiv$ be the smallest right congruence satisfying E0 and E1.*

1. *$\equiv$ is the Nerode equivalence of the language $L(A) = \Lambda_A$ accepted by A.*
2. *For every $w \in \Sigma^*$, either $w \equiv 0$, or there is a unique $i \in P$ such that $w \equiv 1^i$. Moreover, $0 \not\equiv 1^i \not\equiv 1^j$, for $i, j \geq 0$ and $i \neq j$.*
3. *The equivalence classes of $\equiv$ are the languages $D0\Sigma^*$ and Λ_n for $n \geq 0$.*

Proof. Let $\equiv_L$ be the Nerode equivalence of the language $L = L(A)$ of automaton A of Fig. 1. Clearly, $\equiv_L$ satisfies E. Also, since A is reduced, the states of A correspond to the $\equiv_L$-classes via the mapping

$$i \rightarrow [1^i]_L, \text{ for } i \geq 0, \text{ and } \infty \rightarrow [0]_L,$$

in a one-to-one fashion. By definition of A, $\delta(q, 10) = q$, for all $q \in Q$, and $\delta(0,0) = \delta(0, 0w)$, for all $w \in \Sigma^*$. Therefore, $[w10]_L = [w]_L$, and $[0]_L = [0w]_L$. Thus $\equiv_L$ is a right congruence satisfying E0 and E1. Since $\equiv$ is the smallest right congruence satisfying E0 and E1, we must have $\equiv \subseteq \equiv_L$.

By Prop. 4, every word $w \in \Sigma^*$ is $\equiv$-equivalent to either 0, or 1^i for some $i \geq 0$. Note that $0 \not\equiv_L 1^i \not\equiv_L 1^j$. Since $\equiv \subseteq \equiv_L$, we also have $0 \not\equiv 1^i \not\equiv 1^j$. This then proves that $\equiv = \equiv_L$, and thus the first and second claims. The third claim is obvious from the automaton. □

Observe that the transformations described in Prop. 4, together with Prop. 5 provide an effective and efficient procedure (linear in the sum of the lengths of the two words) for deciding whether two words are equivalent. The minimality assumption plays a crucial role in the proof of Prop. 5. Without this assumption one could have $0 \equiv \epsilon$, and this is certainly not the intention in [1], for it would imply that every trace is legal. The minimality assumption is not explicitly stated in [1]. This assumption could be avoided by adding the assertion that $\lambda(0) =$ false.

Our assertion E0, which is not present in [1], serves the purpose of making all illegal traces equivalent. This simplifies the mathematical model without changing the intended meaning. If one wanted to identify several different error conditions, one could use a more refined definition of $\equiv$ on illegal traces.

We now turn our attention to the definition of legality. In the trace-assertion specification method there are only three possible ways to conclude that a word is legal: First, we know by L0 that ϵ is legal. Second, if w is legal, we conclude that $w1$ is legal by L1. Third, if w is legal and $w' \equiv w$, we conclude that w' is legal, by L. Thus, one could construct the set of legal words starting with ϵ and then applying L and L1. This could be done in many different orders. It turns out to be convenient to do it in the following fashion. We start with ϵ, and close it under the equivalence $\equiv$, as permitted by L. This way we obtain the equivalence class $[\epsilon]$. Next we apply L1, obtaining $[\epsilon]1$. We close $[\epsilon]1$ under equivalence, obtaining $[1]$, etc.

Theorem 1. *Let Λ be the set of legal words in the trace-assertion specification. Then $\Lambda = \Lambda_A$.*

Proof. Starting with $\{\epsilon\}$ as the basic set of legal words, we apply L, and then repeatedly L1 followed by L. We count only the number of applications of L1.

With zero applications of L1, we get the equivalence class of the empty word. By Prop. 5, this is the language Λ_0.

Assume now that the set of words constructed using i applications of L1 is the language Λ_i. The next application of L1 results in $\Lambda_i 1$. The application of L to $\Lambda_i 1$ results in the closure of $\Lambda_i 1$ under $\equiv$, yielding Λ_{i+1}.

By Prop. 1, $\Lambda_A = \bigcup_{i=0}^{\infty} \Lambda_i$; hence, $\Lambda = \Lambda_A$. □

An assertion specification is *complete*, if every word in the automaton specification Λ_A is generated in Λ. It is *consistent*[1], if every word generated in Λ is in Λ_A. Thus Theorem 1 states that the assertion specification of the unary counter is complete and consistent with respect to the language specification.

6 Stack Specification by an Automaton

For any integer $z \in Z$, the operation PUSH(z) is denoted simply by z, the POP operation, by p, DEPTH, by d, and TOP, by t.

Each legal state of the stack is represented by the stack contents, that is, by a word q in Z^*. The illegal state is ∞.

Definition 2. *A* stack *is a Mealy automaton $M = (\Sigma, Q, \delta, q_0, F, \Omega, \nu)$, where $\Sigma = \{d, p, t\} \cup Z$, $Q = Z^* \cup \{\infty\}$, $q_0 = \epsilon$, $F = Z^*$, $\Omega = Z$, and δ and ν are defined below. Note that δ is a total function, while ν is a partial function, defined only if $q \in Z^*$ and $a = d$, or $q \in Z^+$ and $a = t$. For all $a \in \Sigma$, $q \in Z^*$, and $z \in Z$, we have:*

- $\delta(\infty, a) = \infty$,
- $\delta(q, z) = qz$,
- $\delta(q, d) = q$, *and* $\nu(q, d) = |q|$,
- $\delta(qz, p) = q$,

[1] The terms *complete* and *consistent* are used in a different sense in [1, 3].

- $\delta(\epsilon, p) = \infty$,
- $\delta(qz, t) = qz$, *and* $\nu(qz, t) = z$,
- $\delta(\epsilon, t) = \infty$.

Note that d never changes the state of the stack, and t changes it only if illegally applied. If t is illegal, the stack goes to state ∞. For each state $q \in Z^*$, that is, for each legal stack state, $\nu(q, d) = |q|$ gives the number of integers on the stack. For each state $q \in Z^+$, that is, for each nonempty stack state, qz, $\nu(qz, t) = z$ gives the top integer on the stack.

The Mealy automaton for the stack is illustrated in Fig. 2. Given state q and input a, if there is no output, the transition from q under a is labelled by a. If there is an output b, the transition is labelled by the pair (a, b). If we treat all the integer inputs together as a PUSH input, then PUSH and p (POP) act exactly as they do in the counter. In the figure, we can only indicate a few typical features of the automaton. The initial state is marked ϵ. There is one transition from this state for each of d, p, and t. There is also one transition from ϵ for each integer; hence the number of such transitions is infinite. If we apply input 3, representing PUSH(3), the automaton moves to state (3). If we then apply 5, the state becomes (3, 5), etc.

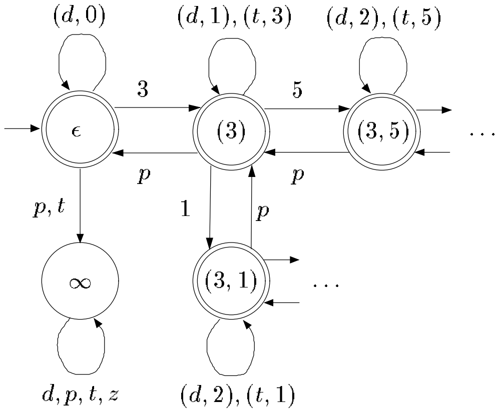

Fig. 2. Mealy automaton of stack

The legal traces of the stack are now specified by $\Lambda_M = L(M)$. Note that if w is in Λ_M, then so is each of its prefixes.

Let $M_q = (\Sigma, Q, \delta, \epsilon, \{q\}, \Omega, \nu)$ be the automaton obtained from M by changing the set of accepting states to $\{q\}$, where $q \in Z^*$. Also, denote $L(M_q)$ by Λ_q. Then

$$\Lambda_M = \bigcup_{q \in Z^*} \Lambda_q.$$

Let Γ be the set of output-returning operations (V-functions) in Σ; then $\Sigma \setminus \Gamma$ is the set of O-functions. For any $w \in \Sigma^*$ and $a \in \Gamma$, the partial function

$\nu(wa)$ is defined by

$$\nu(wa) = \nu(\delta(\epsilon, w), a).$$

We use the convention that $\nu(wa) = \nu(w'a)$ means that either $\nu(wa)$ and $\nu(w'a)$ are both undefined, or are both defined and have the same value. The equivalence relation $\equiv_M$ on Σ^* is defined as follows: $w \equiv_M w'$ if and only if,

$$\text{for all } u \in \Sigma^*, wu \in \Lambda_M \Leftrightarrow w'u \in \Lambda_M \ \wedge \ \text{ for all } a \in \Gamma,\ \nu(wua) = \nu(w'ua).$$

A Mealy automaton M is *reduced* with respect to the equivalence $\equiv_M$ if and only if

$$w \equiv_M w' \Rightarrow \delta(\epsilon, w) = \delta(\epsilon, w').$$

Proposition 6. *The stack automaton M of Fig. 2 is reduced.*

Proof. First, state ∞ 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 $q_{i+1} \cdots q_n$. Then $q_i \neq q'_i$. Now q and q' are distinguishable by $p^{n-i}t$. □

We now characterize the language accepted by M. Let $||w|| = |w|_Z - |w|_p$, where $|w|_Z$ is the number of integers appearing in w.

Proposition 7.

$$L(M) = \{w \mid w = uv \Rightarrow ||u|| \geq 0, \text{ and } w = utv \Rightarrow ||u|| > 0\}.$$

Proof. Since this result is not used, we leave its verification to the reader. □

As for the counter, this result could be used as a type of trace-assertion specification for the set of legal traces.

7 Verification of the Stack

In our notation, the stack specification becomes as follows. Here w is an arbitrary word of Σ^*.

Syntax:

$p, z : \langle\text{stack}\rangle \to \langle\text{stack}\rangle,\quad$ for all $z \in Z$,
$d, t : \langle\text{stack}\rangle \to \langle\text{integer}\rangle$.

Legality:

L: $w \equiv w' \Rightarrow \lambda(w) = \lambda(w')$,
L0: $\lambda(\epsilon) = \text{true}$,
L1: $\lambda(w) \Rightarrow \lambda(wz),\quad$ for all $z \in Z$,

Equivalence: The equivalence relation is the smallest equivalence satisfying

E: $w \equiv w' \Rightarrow wu \equiv w'u,\quad$ for all $u \in \Sigma^*$,
E0: $p \equiv pw \equiv t$,
E1: $w \equiv wzp$,
E2: $w \equiv wd$,
E3: $wzt \equiv wz$.

Values: Here we denote the output function as θ to distinguish it from the output function ν of the automaton. As before, let Γ be the subset of Σ consisting of the V-functions. For all $w \in \Sigma^*$, we have

V: $\lambda(w) \wedge \lambda(wa) \wedge w \equiv w' \Rightarrow \theta(wa) = \theta(w'a)$, for any $a \in \Gamma$,
V0: $\theta(d) = 0$,
V1: $\lambda(w) \Rightarrow \theta(wzd) = 1 + \theta(wd)$,
V2: $\lambda(w) \Rightarrow \theta(wzt) = z$.

We first show that every word $w \in \Sigma^*$ has a canonical representative.

Proposition 8. *For every $w \in \Sigma^*$, either $w \equiv p$ or there exists $q \in Z^*$ such that $w \equiv q$.*

Proof. We apply the following $\equiv$-equivalence-preserving transformations to w, as explained below. In all cases $u, v \in \Sigma^*$.

0. If $w = pu$ or $w = tu$, then $w \rightarrow p$.
1. If $w = uzpv$, then $w \rightarrow uv$.
2. If $w = udv$, then $w \rightarrow uv$.
3. If $w = uztv$ and then $w \rightarrow uzv$.

Transformation 0 preserves $\equiv$ by E0 and E. Transformation 1 preserves $\equiv$ because $uzp \equiv u$ by E1, and $uzpv \equiv uv$ by E. Transformation 2 preserves $\equiv$ because $ud \equiv u$ by E2, and $udv \equiv uv$ by E. Transformation 3, preserves $\equiv$ because $uzt \equiv uz$ by E3, and $uztv \equiv uzv$ by E.

To be more precise, we proceed as follows:

(a) By repeatedly applying transformation 2 we remove all occurrences of d. From now on assume that w does not contain d.
(b) If $w = pu$ or $w = tu$, then $w \rightarrow p$ by transformation 0.
(c) By repeatedly applying transformation 1, we remove all occurrences of zp. Note that removing an occurrence of zp may create an occurrence of zt (but in a shorter word). For example, $zzpt \rightarrow zt$. Also, using this transformation may introduce a word beginning with t. For example, $zpt \rightarrow t$.
(d) By repeatedly applying transformation 3, we reduce all occurrences of zt to z. Note that reducing zt to z, may create an occurrence of zp (but in a shorter word). For example, $ztp \rightarrow zp$.
Apply steps (b)–(d) repeatedly until the word is p, or there are no more occurrences of zp and zt. This leaves a word in Z^* as the only possibility.

Note that no transformation is applicable if $w = p$, or $w \in Z^*$. Hence our claim holds. □

Proposition 9. *Let $\equiv$ be the smallest right congruence satisfying E0–E3. Then*

1. $\equiv\, = \,\equiv_M$.
2. *For every $w \in \Sigma^*$, either $w \equiv p$, or there is a unique $q \in Z^*$ such that $w \equiv q$. Moreover, $p \not\equiv q \not\equiv q'$, for $q, q' \in Z^*$ and $q \neq q'$.*
3. *The equivalence classes of $\equiv$ are precisely the languages $\Lambda_\epsilon(p \cup t)\Sigma^*$ and Λ_q for $q \in Z^*$.*

Proof. Since M is reduced with respect to $\equiv_M$, the states of M correspond to the $\equiv_M$-classes via the mapping

$$q \to [q]_M \text{ for } q \in Z^*, \text{ and } \infty \to [p]_M,$$

in a one-to-one fashion. By definition of M,

- $\delta(\epsilon, p) = \delta(\epsilon, pw) = \delta(\epsilon, t)$, for all $w \in \Sigma^*$,
- $\delta(q, zp) = q$, for all $q \in Q$,
- $\delta(q, d) = q$, for all $q \in Q$, and
- For all $q \in Z^*$, $qz = \delta(qz, t) = \delta(\delta(q, z), t) = \delta(q, zt)$. Since $\delta(q, z) = qz$, we have $\delta(q, zt) = \delta(q, z)$.

Therefore,

- $[p]_M = [pw]_M = [t]_M$, for all $w \in \Sigma^*$,
- $[wzp]_M = [w]_M$, for all $w \in \Sigma^*$,
- $[wd]_M = [w]_M$, for all $w \in \Sigma^*$, and
- $[wzt]_M = [wz]_M$, for all $w \in Z^*$.

Thus $\equiv_M$ is a right congruence satisfying E0–E3. Since $\equiv$ is the smallest right congruence satisfying E0–E3, we must have $\equiv\, \subseteq\, \equiv_M$.

By Prop. 8, every word $w \in \Sigma^*$ is $\equiv$-equivalent to either p, or q for some $q \in Z^*$. Note that $p \not\equiv_M q \not\equiv_M q'$, if $q \neq q'$. Since $\equiv\, \subseteq\, \equiv_M$, we also have $p \not\equiv q \not\equiv q'$. This then proves that $\equiv\, = \,\equiv_M$, and thus the first and second claims. The third claim is obvious from the automaton. □

As in the case of the counter, the transformations of Prop. 8 constitute an efficient algorithm for deciding the equivalence of two words.

Theorem 2. *Let Λ be the set of legal words in the trace-assertion specification. Then $\Lambda = \Lambda_M$.*

Proof. Given a set of legal words, we can find new legal words by using rules L1 and L. The set of words obtained using L alone is the equivalence class of the empty word. By Prop. 9, this is Λ_ϵ.

Assume now that the set of words constructed using i applications of L1 with integers $z_1, \ldots, z_i$ and L is the language Λ_q, where $q = z_1 \ldots z_i$. The $(i+1)$st application of L1 results in $\Lambda_q z_{i+1}$. The application of L to $\Lambda_q z_{i+1}$ results in the closure of $\Lambda_q z_{i+1}$ under $\equiv$, yielding $\Lambda_{qz_{i+1}}$.

Altogether, $\Lambda = \bigcup_{q \in Q} \Lambda_q$. As we have observed in Section 6, $\Lambda_M = \bigcup_{q \in Q} \Lambda_q$. Hence $\Lambda_M = \Lambda$. □

To complete the verification of the stack specification, we prove that the output produced in the trace-assertion specification is always the same as that produced by the automaton.

Proposition 10. *For every $w \in Z^*$, $\nu(w, d) = \theta(wd)$.*

Proof. First, $\nu(w, d) = |w|$, by definition of M. For θ, we proceed by induction on the length of w. If $|w| = 0$, then $\theta(wd) = \theta(d) = 0$, by V0. Since $|\epsilon| = 0$, the claim holds. Assume now that $\theta(wd) = |w|$ for all w of length less than or equal to n, and consider wzd, where $|w| = n$, and $z \in Z$. Since $\lambda(w) =$ true if $w \in Z^*$, by Theorem 2, V1 is applicable. Also, $\lambda(wz) =$ true by L1, and $\lambda(wzd) =$ true by E2 and L. By V1, $\theta(wzd) = 1 + \theta(wd) = 1 + n = |wz|$. Hence the induction goes through and the claim follows. □

Proposition 11. *For every $w \in Z^+$, $\nu(w, t) = \theta(wt)$.*

Proof. In the automaton M, we have $\nu(wz, t) = z$, while in the trace-assertion specification, $\theta(wzt) = z$ by V2, and the claim follows. □

Theorem 3. *The output values of the trace-assertion specification of the stack are correct with respect to those of the stack automaton.*

Proof. We have shown in Props. 10 and 11 that the outputs agree for all canonical traces. Suppose now that w is any legal trace, and $\hat{w}$ is its canonical representative. Then w and $\hat{w}$ take M to the same state, and will produce the same output when d is applied. By V, $w \equiv \hat{w} \Rightarrow \theta(wd) = \theta(\hat{w}d)$. The same type of reasoning applies if the input is t. □

8 Conclusions

We have used a formal method for studying trace-assertion specifications. We have shown that there are flaws in the earlier definitions of the Bartussek-Parnas model. We have corrected these flaws, and proved the correctness of the trace-assertion method for a counter and a stack. We have shown that, in the cases of the counter and stack, canonical representatives of traces can be derived directly from the trace specification in a natural way, and need not be artificially introduced. In view of these results, the original method of Bartussek and Parnas (with the indicated corrections) deserves to be revisited.

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

References

1. Bartussek, W. and Parnas, D.: Using Assertions About Traces to Write Abstract Specifications for Software Modules. Report No. TR77-012, University of North Carolina at Chapel Hill, December (1977) 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. Berstel, J.: Transductions and Context-Free Languages. B. G. Teubner, Stuttgart (1979)
5. Hopcroft, J. E. and Ullman, J. D.: Introduction to Automata Theory, Languages and Computations. Addison-Wesley (1979)
6. 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/
7. 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)
8. Salomaa, A.: Theory of Automata. Pergamon Press, Oxford (1969)
9. Wang, Y.: Formal and Abstract Software Module Specifications — A Survey. Tech. Rept. 91–307, Computing and Information Science, Queen's University, Kingston, ON, (1991)
10. 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
11. Wood, D.: Theory of Computation. Harper and Row (1987)