Specifying Module Interfaces with Finitely Defined Automata

RYSZARD JANICKI*
McMaster University
Department of Computing and Software
Hamilton, Ontario, L8S 4K1
CANADA

Abstract: The use of automata as a specification tool is analyzed. Trace Assertion Method (TAM) is used as an example. TAM is a formal method for specifying module interfaces. It treats the module as a black box, and was designed as an alternative to an algebraic specification technique [1], and became quite popular in the software industry [13]. Finiteness and concurrency issues are discussed.

Key–Words: trace assertion, automata, module interfaces, formal specification

1 Introduction

Let us start with the following fundamental question: "What is the basic tool used to model systems?". In classical science and engineering the answer would be *equations*. For instance to model the relationship between acceleration and and distance we write the equation

$$ds = vdt = atdt \Rightarrow \frac{ds}{dt} = st \Rightarrow$$

$$s = \int_0^t atdt = \frac{1}{2}dt^2.$$

In Computer Science and Software Engineering the answer is not so obvious, but I believe the answer would most likely be *automata (state machines, transition systems)* or *algebraic equations (fixpoints)*. The equivalence of automata (state machines, transition systems) and algebraic equation (fixpoints) approaches will be discussed in a sequel, however the automata (state machines, transition systems) approach seems to be superior at least from the point of view of practical applications. The concept of an automaton (state machine) is very well understood by most of engineering and computer science graduates. However, there are still many unnecessary but frequent assumptions and conventions that prevent from even more frequent use of this concept. Most of the unnecessary assumptions and conventions belong to the following list:

- the set of states is finite,
- the set of transitions (actions, symbols) is finite.
- there is only one transition function (relation).
- graphical representation is a major motivation.
- the states and transitions (actions) are considered as abstract entities related only by a transition function.
- the transition function is an abstract entity not related to states and transitions.
- the use of expressions as a specification tool is marginal.

We do not define the function "add 3" as $f : Int \rightarrow Int$ and $f(1) = 4$, $f(2) = 5$, $f(3) = 6$, etc., we just write and *expression* $f(n) = n + 3$. However when it comes to automata, the expressions are not frequently used to define information flow.

Let us consider the following popular example: a bounded stack of integers, with the maximal length equal to $size$. It could be easily specified as the following automaton:

States: sequences of integers as $1.3.2.3$, 56.1, $\epsilon =$ empty sequence, etc.,
Actions/Alphabet: $push(i)$, pop, $top{:}i$,
Transition Function: $\delta : States \times States \rightarrow States$, where

$$\delta(x, push(i) = \begin{cases} x.i & length(x) < size \\ x & length(x) = size \end{cases}$$

$$\delta(x, pop) = \begin{cases} y & x = y.i \\ \epsilon & x = \epsilon \end{cases}$$

$$\delta(x, top{:}i) = \begin{cases} x & x = y.i \\ nil & x = \epsilon \end{cases}$$

The automaton is *interpreted*, as states and actions are no longer abstract, it is *finitely* defined, since

*Partially supported by NSERC of Canada Grant

the states, actions, and the transition function are completely defined by *a finite set of expressions and equations*. Note that all the equations are *closed* and *explicit*.

As an another example let us consider the ADT of Integers:

States: integers,
Actions/Alphabet: integers,
Transition Functions (four):
$\delta_+, \delta_-, \delta_\times, \delta_/ : States \times States \rightarrow States$, and
$\delta_+(i,j) = i + j$, $\delta_-(i,j) = i - j$
$\delta_\times(i,j) = i \times j$, $\delta_/(i,j) =$ if $j \neq 0$ then $i$div3 else nil.

It is a well established fact that state machine (not necessary finite) models and algebraic models are equivalent ([4, 5]). This relationship differs for different machines and algebras, but the general idea of the relationship may be illustrated as follows:

$$\underbrace{\delta(p,a) = q}_{\textit{state machine}} \quad \Leftrightarrow \quad \underbrace{a(p) = q}_{\textit{algebra}},$$

where δ is a transition function of a state machine with a as a function name, and $a(p)$ is a function named a applied to p.

Often automata models are better suited for specifying and analyzing concrete software systems, while algebraic models are better suited for defining more abstract and general theories.

Let consider the example of a stack again. The most popular algebraic specification of this module looks as follows:

$$pop(push(s,a) = s$$
$$top(push(s,a) = a$$

The Algebraic Specification have the following properties:

- Implicit open equations.
- Elegant solutions for "regular cases",
- Problems with "irregular cases", as "stack is bounded", when it is full then $push$ may either do nothing or deletes the bottom.
- Equations are not obvious for complex cases.
- Non-determinism and concurrency make the model very complex.

The properties of *Automata* base specifications are complimentary to the Algebraic Specification properties:

- Explicit closed equations.
- Usually uglier solutions for "regular cases".
- "Irregular cases" are handled easily.
- Equations can be obtained naturally for complex cases.
- Non-determinism and concurrency only slightly increase the complexity of a model.

The differences described above were the major motivation for the introduction of *the trace assertion method* (see [13]). The trace assertion method was first formulated by Bartussek and Parnas in [1], as a possible answer for some of the problems with algebraic specifications [19], like specifying a bounded stack (bounded modules in general). It also can avoid the problem of overspecification in model-oriented specifications, e.g. [13]. Since its introduction the method has undergone many modifications [8, 15, 13]. In recent years, there has been an increased interest in the trace assertion method [13]. Despite many important industry applications (see [9]), solid mathematical foundations of trace assertion method have been provided only very recently, see [3, 13, 12].

The trace assertion method is based on the following postulates:

1. *Information hiding* [16, 17] is a fundamental principle for specification, so we describe only those features of a module that are externally observable;
2. *Sequences* are simple and powerful tool for specifying abstract *objects*;
3. *Explicit equations* are preferable over *implicit equations* like those of algebraic specifications;
4. *State machines* are simple and powerful tools for specifying *modules*.

As stated above, the fundamental difference between algebraic specification and the trace assertion method is that algebraic specification supports *implicit equations*, while trace assertion method uses *explicit equations* only.

The areas of applications for algebraic specifications are different than for the trace assertion method. The algebraic specification is better suited for defining abstract data types in programming languages (as SML, LARCH, etc., see [19]). The trace assertion method is better suited for specifying complex interface modules as for instance communication protocols [8]. A very wide bibliography concerning the Trace Assertion Method can be found in [13].

2 Introductory Examples

We shall consider the following simple modules: Queue, Drunk Queue, Very Drunk Queue, Concurrent Queue and Concurrent Drunk Queue. The Queue module provides four access programs: *insert*(i) - which inserts an integer i to the rear of the queue, *remove* - which takes no argument and removes the first element of the queue,
front - which takes no argument and returns the value of the first element of the queue,

rear - which takes no argument and returns the value of the last element of the queue.

Since a trace specification describes only those features of a module that are externally observable, the question of what an atomic observation is arises. Following [13], we assume that an atomic observation is a pair

$$(ap(ar), vr)$$

where $ap = access\_program$, $ar = arguments$, and $vr = value\_returned$, written as

$$ap(ar){:}vr.$$

No argument and no returned value is represented by nil, however we also adopt a convention of omitting nil, in particular as arguments. Hence, the Queue module has the following atomic observations, called *call-responses*: $insert(i) : nil$, $remove(nil) : nil$, $front(nil){:}a$, $rear(nil){:}b$, or, when nil's are omitted: $insert(i)$, $remove$, $front{:}a$, $rear{:}b$, where a is the value of first element of the queue, and b is the value of the last element in the queue.

Intuitively, a state of the queue is determined by a finite sequence of integers, the last element of the sequence represents the rear of the queue, and the first represents the beginning of the queue. Note that every sequence of properly used access programs leads to exactly one state. For instance *insert*(4).*insert*(1).*remove*.*insert*(7) and *insert*(1).*insert*(7) both lead to the state $\langle 1, 7\rangle$. They could be seen as equivalent and we can choose for instance the trace *insert*(1).*insert*(7) as a *canonical trace* representing the state $\langle 1, 7\rangle$.

Module Drunk Queue is the same as Queue except that the access program *remove* behaves differently, namely: if the length of the queue is one it removes the first element; and if it is greater than one it removes either the first element or the first *two* elements of the queue. Now the trace *insert*(4).*insert*(1).*remove*.*insert*(7) may lead to two states: $\langle 1, 7\rangle$ or $\langle 7\rangle$. However, each state is unambiguously described by an appropriate trace built from *insert* calls.

The Concurrent Queue has the same access programs as Queue, but *simultaneous* calls are allowed, for instance if a queue is not empty, a simultaneous call of *insert*(i) and *remove* is allowed, as well as a simultaneous call of *insert*(i) and *front*, or *remove* and *rear*. Simultaneous calls might be represented by *steps* like $\{insert(5), remove\}$, and it is more convenient to use *step-traces* to represent the observations. For instance the step-trace $\{insert(1)\}.\{insert(5), front{:}1\}.\{remove, rear{:}5\}$ leads to the state $\langle 5\rangle$.

The Concurrent Drunk Queue has "drunk" *remove* and allows simultaneous calls.

3 The Model

3.1 Type of Concurrency and Alphabet

We assume that executions (observations) of concurrent behaviours can fully be modeled by *step-sequences* (or, equivalently stratified posets). This means we assume simultaneity is observable and, when restricted to single concurrent history, it is also transitive. We also assume the a possibility of simultaneous execution of a and b implies a possibility of execution in the order a followed by b, and in the order b followed by a (see for instance [10] for discussion of various models of concurrency). We are fully aware of the restrictions imposed by the model we have chosen. Its basic advantage is simplicity, and yet ability to model a wide spectrum of systems.

What formally constitutes an alphabet from which the traces are built? Let f be the name of an access program and let $input(f)$ and $output(f)$ be the sets of possible argument and result values. The *signature* $sig(f)$ is the triple:

$$sig(f) = (f, input(f), output(f)).$$

We assume that neither $input(f)$ nor $output(f)$ are empty by having $nil \in input(f)$ and $nil \in output(f)$ as default. For example:

$$sig(\textit{insert}) = (\textit{insert}, integer, \{nil\}),$$
$$sig(\textit{remove}) = (\textit{remove}, \{nil\}, \{nil\}),$$
$$sig(\textit{front}) = (\textit{front}, \{nil\}, integer),$$
$$sig(\textit{rear}) = (\textit{rear}, \{nil\}, integer).$$

For a finite set E of access program names, the *signature* $sig(E)$ is the set of all signatures of $f \in E$:

$$sig(E) = \{sig(f) \mid f \in E\}.$$

Given E, the *call-response alphabet* Δ_E is the set of all possible triples, written $f(x){:}g$ of access program names, arguments, and return values:

$$\Delta_E = \{f(x){:}g \mid f \in E, x \in input(f), y \in output(f)\}.$$

We adopt the convention of omitting nil in signatures. For example, for the queue modules we have $E = \{\textit{insert}, \textit{remove}, \textit{front}, \textit{rear}\}$ and:

$$\Delta_E = \{\textit{insert}(i) \mid i \in integer\} \cup \{\textit{front}{:}i \mid i \in integer\} \cup \{\textit{rear}{:}i \mid i \in integer\} \cup \{\textit{remove}\}.$$

For a given set E of access program names, we also define the *call alphabet* Σ_E and the *response alphabet* $\mathcal{O}_E$:

$$\Sigma_E = \{f(x) \mid f \in E, x \in input(f)\},$$
$$\mathcal{O}_E = \{d \mid \exists f \in E.\ d \in output(f)\}.$$

Note that the sequences and step-sequences of call-response event occurrences are what is really observed.

3.2 Trace Assertion Specification

For every set X, let $\mathcal{S}(X) = \{A \mid \emptyset \neq A \subseteq X \wedge A \text{ is finite}\}$. Elements of $\mathcal{S}(X)$ will be called

steps, while elements of $\mathcal{S}(X)^*$ are called *step-sequences*. For instance, if $X = \{a, b, c\}$, then $\{a, b\}.\{b\}.\{a, b, c\} \in \mathcal{S}(X)^*$ is a step-sequence. Traditionally λ denotes the empty step-sequence. If it is a set of calls a *step-trace* is $X = \Delta_E$ for some E.

For every set X, let $Rel(X) = \{R \mid R \subseteq X \times X\}$, and for every symmetric $R \in Rel(X)$, let $cliques(R) \subseteq \mathcal{S}(X)$, the set of all cliques of R, be the set defined as follows: for every $x \in X$, $\{x\} \in cliques(R)$ and for every finite $A = \{x_1, ..., x_k\} \subseteq X$, $A \in cliques(C)$ iff $(x_i, x_j) \in R$ for $i \neq j$.

In principle a Trace Assertion Specification is an automaton with call-response events as an alphabet (possibly infinite), and some sequences of call-response events (traces) as states (possibly infinite). However, the automaton is *finitely defined*, in the sense that the number of *explicit* equations that define the elements of the alphabet, states and transition function is finite. For practical applications is is also important that the number of these equations is small and that they are relatively simple. The fact that the expressions are explicit (as opposed to algebraic specification where implicit equations are more natural) is extremely important from the application viewpoint, even though it is not very significant as far as the theory is concerned (see [13] for details).

Formally a *Concurrent Full Trace Assertion Specification* is a tuple:

$$CFTA = (sig(E), \mathcal{C}, \delta, \delta_c, \mathcal{K}, enabled, t_0),$$

where:

- E is the set of names of *system calls*, $|E| < \infty$,
- $sig(E)$ is the *signature* defined by E,
- $\mathcal{C} \subseteq \mathcal{S}(\Delta_E)^*$ is the set of *canonical step-traces* (state descriptors),
- $\delta : \mathcal{C} \times \Delta_E \to 2^{\mathcal{C}}$ is the *sequential transition function*, and $\delta^* : \mathcal{C} \times \Delta_E^* \to 2^{\mathcal{C}}$ is a standard extension of δ onto Δ_E^* (see [5]). In general an automaton that is a frame for CTA is non-deterministic, so the range of δ is defined as $2^{\mathcal{C}}$, see [13],
- $\delta_c : \mathcal{C} \times \mathcal{S}(\Delta_E) \to 2^{\mathcal{C}}$ is the *concurrent transition function*, and $\delta_c^* : \mathcal{C} \times \mathcal{S}(\Delta_E)^* \to 2^{\mathcal{C}}$ is a standard extension of δ onto $\mathcal{S}(\Delta_E)^*$,
- $\mathcal{K} \subseteq \mathcal{C} \times \Sigma_E$ is a *competence* set, if $(c, \alpha) \notin \mathcal{K}$, then applying the system call α at the state c is an erroneous/exceptional behaviour, as for instance the *remove* call at empty queue,
- $enabled : \mathcal{C} \to 2^{\mathcal{S}(\Delta_E)}$ is the mapping that defines concurrency; it states what *steps* are *enabled* at each state (canonical trace),
- $t_0 \in \mathcal{C}$ is the *initial* (canonical) *state*,

and the following conditions are satisfied:

1. for all $c \in \mathcal{C}$, $\delta_c^*(t_0, c) = \{c\}$,
2. for all $c \in \mathcal{C}$, and all $S_1, S_2 \in \mathcal{S}(\Delta_E)$,
3. $S_1 \subseteq S_2 \in enabled(c) \Rightarrow S_1 \in enebled(c)$,
4. for all $c \in \mathcal{C}$, if $\{\alpha, \beta\} \in enabled(c)$, then $\delta^*(c, \alpha.\beta) = \delta^*(c, \beta.\alpha)$,
5. for all $c \in \mathcal{C}$, and all $A \in \mathcal{S}(\Delta_E)$, if $A = \{\alpha_1, ..., \alpha_k\} \in enabled(c)$ then $\delta_c(c, A) = \delta^*(c, \alpha_1. \ ... \ .\alpha_k)$, otherwise $\delta_c(c, A) = \emptyset$,
6. for all $c \in \mathcal{C}$ and all $a{:}d \in \Delta_E$, if there exists $A \in enabled(c)$ such that $a{:}d \in A$ and $|A| \geq 2$ then $(c, a) \in \mathcal{K}$.
7. for all $c \in \mathcal{C}$ and all $a \in \Sigma_E$ there exists $d \in \mathcal{O}_E$ such that $\delta(c, a{:}d) \neq \emptyset$,
8. for all $c \in \mathcal{C}$ and all $\alpha \in \Delta_E$, $\delta(c, \alpha) \neq \emptyset \iff \{\alpha\} \in enabled(c)$.

Condition (1) guarantees that the states are correctly and uniquely defined by canonical traces. The second condition says that every non-empty subset of an enabled step is also an enabled step at the given state c. This means we *do not* enforce maximal concurrency (see [11]). Condition (3) enforces the rule that simultaneous executions of $\{\alpha, \beta\}$ implies that both α followed by β and β followed by α are possible orderings. Condition (4) defines the concurrent transition δ_c by the sequential transition δ. As a matter of fact, the concurrent transition function δ_c is redundant, since it is fully described by δ and $enabled$, however it makes theoretical considerations and definitions easier and more readable. However in concrete examples it is usually omitted (see an example in Figure 1). The fifth condition states that concurrent activity is restricted to normal non-erroneous behaviour. Any exceptional activity must be sequential. This follows from the suggestions of practitioners who recommend not to mix concurrency with erroneous behaviour, since the results might become difficult to handle. Condition (6) is based on the observation that we cannot practically forbid the use of system calls "illegally" (there is always a possibility that somebody will try to apply *remove* to an empty queue), so the specification should be able to handle such cases. The last condition states that δ and $enabled$ do not contradict eachother.

The functions δ and δ_c can be decomposed into δ^N, δ^{err}, and δ_c^N, δ_c^{err}, as follows. For all $c \in \mathcal{C}$ and all $a{:}d \in \Delta_E$, we have:

$$\delta^N(c, a{:}d) = \begin{cases} \delta(c, a{:}d) & (c, a) \in \mathcal{K} \\ \emptyset & (c, a) \notin \mathcal{K} \end{cases}$$

and

$$\delta^{err}(c, a{:}d) = \begin{cases} \emptyset & (c, a) \in \mathcal{K} \\ \delta(c, a{:}d) & (c, a) \notin \mathcal{K} \end{cases} .$$

Syntax of Access Programs

Name	Argument	Value	Action-response Form	Full Action-response Form
$Front$		$integer$	$Front{:}d$	$Front{:}d$
$Rear$		$integer$	$Rear{:}d$	$Rear{:}d$
$Insert$	*integer*		$Insert(a)$	$Insert(a){:}nil$
$Remove$			$Remove$	$Remove{:}nil$

Canonical Step-traces

t is *canonical* $\iff$ $(t = \lambda \vee t = \{Insert(a_1)\}. \ldots .\{Insert(a_k)\})$,
where $1 \leq k \leq size$.
$t_0 = \lambda$, i.e. empty step-sequence.

Enabled

if $c = \lambda$ **then** $enabled(c) = \{\ \{Insert(x)\}\ \mid x$ is an integer$\}$.
if $c = \{Insert(a)\}.t_1.\{Insert(b)\} \wedge |c| = size$ **then**
$enabled(c) = \{\ \{Remove\}, \{Front{:}a\}, \{Rear{:}b\}, \{Rear{:}b, Remove\}\ \}$.
if $c = \{Insert(a)\}$ **then**
$enabled(c) = \{\ \{Remove\}, \{Front{:}a\}, \{Rear{:}a\}\ \} \cup$
$\{\ \{Insert(x)\}\ \mid x$ is an integer$\} \cup$
$\{\ \{Insert(x), Remove\}\ \mid x$ is an integer$\} \cup$
$\{\ \{Insert(x), Front{:}a\}\ \mid x$ is an integer$\}$.
if $c = \{Insert(a)\}.t_1.\{Insert(b)\} \wedge |c| < size$ **then** $enabled(c) =$
$\{\{Remove\}, \{Front{:}a\}, \{Rear{:}b\}, \{Rear{:}b, Remove\}, \{Front{:}a, Rear{:}b\}\}$
$\cup \{\ \{Insert(x)\}\ \mid x$ is an integer$\} \cup \{\ \{Insert(x), Remove\}\ \mid x$ is an integer$\} \cup$
$\{\ \{Insert(x), Front{:}a\}\ \mid x$ is an integer$\}$.

Trace Assertions

$\delta(t, \{Front{:}d\}) =$

Condition	Trace Patterns	Result
	$t = \{Insert(d)\}.t_1$	$\{t\}$
% $d = nil$	$t = \varepsilon$	$\{\lambda\}$

$\delta(t, \{Rear{:}d\}) =$

Condition	Trace Patterns	Result
	$t = t_1.\{Insert(d)\}$	$\{t\}$
% $d = nil$	$t = \varepsilon$	$\{\lambda\}$

$\delta(t, \{Insert(a)\}) =$

Condition	Result
$length(t) < size$	$\{\ t.\{Insert(a)\}\ \}$
% $length(t) = size$	$\{t\}$

$\delta(t, \{Remove\}) =$

Trace Patterns	Result
$t = \{Insert(b)\}.t_1$	$\{t_1\}$
% $t = \varepsilon$	$\{\lambda\}$

Dictionary $size$: the size of the queue
$length(t)$: the length of the trace t

Figure 1: Full Trace Assertion Specification for Concurrent Bounded Queue Module

The conditions (4) and (5) guarantee that $\delta_c^N(c, \{\alpha_1, ..., \alpha_k\}) = \delta^{N*}(c, \alpha_1. \ldots \alpha_k)$, and if $A \in enabled(c)$ and $\delta_c^{err}(c, A) \neq \emptyset$, then $A = \{\alpha\}$ is a singleton, and $\delta_c^{err}(c, \{\alpha\}) = \delta^{err}(c, \alpha)$.

Lemma 1

1. $\delta = \delta^N \cup \delta^{err}$ *and* $\delta^N \cap \delta^{err} = \emptyset$,
2. $\delta_c = \delta_c^N \cup \delta_c^{err}$ *and* $\delta_c^N \cap \delta_c^{err} = \emptyset$. □

The functions δ^N, δ_c^N are called *normal* transition and *normal concurrent* transition functions, while the function δ^{err} is called an *exceptional* transition function. Due to the condition (5) the function δ_c^{err} is of little use. The concurrent full trace assertion specification $CFTA$ restricted to the function δ^N is called *concurrent trace assertion specification*, denoted CTA, while $CFTA$ restricted to δ^{err} is called and *enhancement* of CTA and denoted ETA. Lemma 1 allows us to write (informally, but correctly), $CFTA = CTA + ETA$. For concrete examples, CTA (i.e. the functions δ^N, δ_c^N) should be *specified first*, and an enhancement should be added later. Lemma 1 and condition (7) guarantee that such an approach is sound.

The enhancement ETA is called *plain* if $\delta^{err}(c, \alpha) \neq \emptyset$ implies there are c_1 and α_1 such that $\delta^N(c, \alpha_1) \neq \emptyset$ and $\delta^N(c_1, \alpha) \neq \emptyset$. Non-plain enhancement means that there are some special error recovery states and a separate error recovery procedure ([13]). Our example in Figure 1 has a plain enhancement.

We say that $CFTA$ is *deterministic* iff for all $c \in \mathcal{C}$ and all $\alpha \in \Delta_E$, $|\delta(c, \alpha)| \leq 1$. Note that this implies $|\delta_c(c, A)| \leq 1$, for every state c and step A. From the examples introduced in Section 2, Queue and Concurrent Queue are deterministic, the remaining are not-deterministic. The concept of determinism defined above corresponds to the concept of determinism used in automata theory (see [13]).

For a given $CFTA$, let the function
$sim : \mathcal{C} \rightarrow Rel(\{\alpha \mid \{\alpha\} \in enabled(c)\})$
be defined as follows:
$(\alpha, \beta) \in sim(c) \iff \{\alpha, \beta\} \in enabled(c)$.
For every c, $sim(c)$ defines *simultaneity* relation at the state c.

Lemma 2

$A \in enabled(c) \iff A \in cliques(sim(c))$. □

From the above lemma and condition (3) of the $CFTA$ definition, it follows that we may equivalently define $CFTA$ as $CFTA = (sig(E), \mathcal{C}, \delta, \mathcal{K}, sim, t_0)$ with appropriate
changes of the constraints (1) - (6). No definition is better than the other. For the theory the definition with δ_c and $enabled$ seems to be better (see [12]), for specifying the concrete examples δ_c is almost never explicitly specified, for some cases using $enabled$ is better, for others sim is better (compare [12]).

Constraints (1) - (6) have to be proven for every concrete example. They are an essential part of a specification, the part which is frequently called an *obligation proof* in software engineering. If the specification is thoroughly thought of, those proofs are usually easy, but they may be labour consuming, if the specification is complex itself. The use of some automatic theorem provers such as *PVS* or *Simplify* is highly recommended [12].

3.3 Specification Format

To be useful in practice, the trace assertion technique must provide some reliable, readable and easy to use specification format. This issue is completely irrelevant from the theoretical standpoint, but very important if the technique is going to be used outside of

academia. The details of a specification format are found in [13, 12]. It uses Parnas Tabular Expressions (see [17, 14] for more details), for simple cases it appears to be self-explainatory (see Figure 1).

The technique described above is illustrated in Figure 1, which presents a Full Trace Assertion Specification for a Concurrent Queue. The symbol "%" in the definition of δ indicates the parts that define δ^{err}, i.e. exceptional behaviour. Figure 1 provides only the first part of a specification. The second one, "the obligation proof" is not provided. It is relatively easy, but somewhat long so it is omitted. An interested reader is referred to [12].

4 Final Comment

The Trace Assertion Method is an example of a *finitely defined interpreted automaton*. Those automata are characterized by

- Finite set of *expressions* defining states.
- Finite set of *expressions* defining actions (transitions).
- Finite set of *equations* defining transition functions.

Other examples of finitely defined interpreted automata are *Sequence Based Software Specifications* [18], popular *SCR automata* [7], and very powerful Gurevich's Evolving Algebras [6] (also known as Abstract State Machines [2]).

References:

[1] W. Bartussek, D.L. Parnas, Using Assertions About Traces to Write Abstract Specifications for Software Modules, *Lecture Notes in Computer Science 65*, Springer 1978, pp.211-236.

[2] E. Börger, R. Stärk, *Abstract State Machines*, Springer 2003.

[3] J. A. Brzozowski, H. Jürgensen, Representation of Semiautomata by Canonical Words and Equivalences, *Int. J. Foundations of Computer Science*, 16, 5, (2005) 831-850.

[4] P. M. Cohn, *Universal Algebra*, D. Reidel 1981.

[5] S. Eilenberg, *Automata, Languages and Machines*, vol A, Academic Press, 1974.

[6] Y. Gurevich, Evolving Algebras 1993: Lipari Guide, in E. Börger, *Specification and Validation Methods*, Oxford University Press, 1995.

[7] C. Heitmeyer, R.D. Jeffords, B.G. Labaw, Automated Consistency Checking of Requirements Specifications, *ACM Trans. Software Eng. and Methodology*, 5,3 (1996) 231-261.

[8] D.M. Hoffman, The Trace Specification of Communication Protocols, *IEEE Transactions on Computers 34*, 12 (1985), pp.1102-1113.

[9] D.M. Hoffman, D.M. Weiss (Eds.), *Collected Papers by David L. Parnas*, Addison-Weslay, 2001.

[10] R. Janicki, M. Koutny, Structure of Concurrency, *Theoretical Computer Science* 112 (1993), 5-52.

[11] R. Janicki, P. E. Lauer, M. Koutny, R. Devillers, Concurrent and Maximally Concurrent Evolution of Non-Sequential Systems, *Theoretical Computer Science* 43 (1986), 213-238.

[12] R. Janicki, Y. Liu, On Trace Assertion Method of Module Interface Specification with Concurrency, *Lecture Notes in Artificial Intelligence* 2005, Springer 2001, pp. 632-641.

[13] R. Janicki, E. Sekerinski, Foundations of the Trace Assertion Method of Module Interface Specification, *IEEE Trans. on Softw. Eng.*, 27, 7 (2001), pp. 577-598.

[14] R. Janicki, A. Wassyng, Tabular Expressions and Their Relational Semantics, *Fundamenta Informaticae* 67, 4 (2005), 343-370.

[15] J. McLean, A Formal Foundations for the Abstract Specification of Software, *Journal of the ACM*, 31,3 (1984), pp. 600-627.

[16] D. Parnas, A Technique for Software Module Specification with Examples, *Comm. of ACM*, 15,5 (1972), pp. 330-336.

[17] D. Parnas, J. Madey, M. Iglewski, Precise Documentation of Well-Structured Programs, *IEEE Trans. on Softw. Eng.*, 20 (1994), pp. 948-976.

[18] S. J. Prowell, J. H. Poore, Foundations of Sequence Based Software Specification, *IEEE Trans. on Softw. Eng.*, 29, 5 (2003), 417-429.

[19] M. Wirsing, Algebraic Specification, in J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science*, Vol 2., Elsevier Science Publ., 1990, pp. 675-788.