Collecting Statistics over Runtime Executions

Bernd Finkbeiner, Sriram Sankaranarayanan and Henny Sipma

Computer Science Department, Stanford University
Stanford, CA. 94305

Abstract

By collecting statistics over runtime executions of a program we can answer complex queries, such as "what is the average number of packet retransmissions" in a communication protocol, or "how often does process P_1 enter the critical section while process P_2 waits" in a mutual exclusion algorithm. We present an extension to linear-time temporal logic that combines the temporal specification with the collection of statistical data. By translating formulas of this language to alternating automata we obtain a simple and efficient query evaluation algorithm. We illustrate our approach with examples and experimental results.

1 Introduction

Runtime verification [13] is a lightweight approach to program safety. Given a trace of a program execution, we report *success* if the trace satisfies the program specification and *failure* if a fault is detected. There are certain limitations to this approach. Liveness properties, for example, can *never* be falsified on a finite trace. In monitoring applications it is often more helpful to be warned about indicators of impending failure, such as the number of packet retransmissions in a network, than about the actual violation.

In this paper, we present an extension to linear-time temporal logic that combines the temporal specification with the collection of statistical data. Instead of *checking* the property "there are only finitely many retransmissions for each packet", which is vacuously true over finite traces, we *evaluate* queries like "what is the average number of retransmissions" or "what is the maximum packet delay", which give a good picture of the current network status.

[1] This research was supported in part by NSF(ITR) grant CCR-01-21403, by NSF grant CCR-99-00984-001, by ARO grant DAAD19-01-1-0723, and by ARPA/AF contracts F33615-00-C-1693 and F33615-99-C-3014.

Our queries are constructed from *experiments*, which form basic observations at individual trace positions, and *aggregate statistics* which combine the results of multiple experiments. This query language is defined in Section 2. We discuss examples from a communication protocol and a mutual exclusion algorithm. Next, we develop an automata-theoretic solution for the evaluation of queries. We introduce *algebraic alternating automata* in Section 4 and discuss their evaluation over traces. The translation of queries to automata is described in Section 5. Section 6 concludes with experimental results from our prototype implementation.

Related Work

Program profiling has a long history, exemplified in popular tools like *gprof* [11]. However, this research has concentrated mostly on certain specific types of data like running time and memory leaks. Our approach can be used to develop flexible profiling tools that evaluate user-defined temporal queries.

Runtime verification with linear-time temporal logic has received a growing attention recently. Examples include the commercial system Temporal Rover [7], a tool that allows the specifications to be embedded in C, C++, Java, Verilog and VHDL programs. Runtime verification algorithms have also been applied in guiding the Java model checker Java PathFinder developed at NASA [12].

Linear-time temporal logic is a widely used formalism for the specification and verification of reactive and concurrent systems [15]. For static analysis, other extensions to quantitative queries have first been studied in the context of real time systems [8,9]. Recent work along the same lines includes [1,5]. Our query language can be seen as a generalization of the logic MinMax Ctl [5].

Alternating automata [6] are a generalization of nondeterministic automata and $\forall$-automata [14]. Because of their succinctness they are an efficient data structure for many problems in specification and verification [19,18]. The algebraic alternating automata we define in Section 4 are inspired by the *extended alternating automata* of [4]. There, extended alternating automata are used for static *Query Checking*, which determines the set of propositional formulas that satisfy a temporal query over a program. Our general framework for using alternating automata for runtime verification was reported in [10]. In this paper we concretize the general approach by providing a query language and a translation from queries to alternating automata.

2 Specifying Runtime Statistics

2.1 Programs, States and Traces

In our framework runtime verification consists of posing queries about program traces. These queries typically contain expressions over program variables, but they do not further depend on the program or its structure, nor does their evaluation. Therefore it is sufficient to formalize only the notions of states and traces.

Let Σ be a *many-sorted* algebraic signature and P be a *program* with a finite set of variables X. Each variable $x \in X$ is assumed to have a fixed sort τ_x. A *query expression* e is a term in the term algebra $\mathcal{T}(\Sigma, X)$. Given a Σ-algebra $\mathcal{V}$, a $\mathcal{V}$-state of a program P is a map $s : X \mapsto \mathcal{V}$ such that each variable $x \in X$ is assigned a value of the right sort in $\mathcal{V}$. We intentionally confuse a state s with the unique homomorphism $s : \mathcal{T}(\Sigma, X) \mapsto \mathcal{V}$ that extends s. Therefore, if e is an expression, $s(e)$ is defined by this homomorphism.

An expression with sort boolean is called an *assertion*. We assume the existence of an *entailment* relation $\models$ such that a state satisfies an assertion ψ, written $s \models \psi$, if and only if ψ is *true* in s, that is $s(\psi)$ has value *true*.

Queries may return a value or they may fail. Therefore we assume that all sorts contain a special element $\perp$ to indicate the failure of a query.

Queries are interpreted over program traces. Formally, a (P-)*trace* σ of length n is a sequence of states $s_0, s_1, \ldots, s_{n-1}$. We write $\sigma(i)$ to denote the state s_i.

2.2 Experiments

Queries over program traces are constructed from *statistical experiments*: expressions that specify a query about the trace at a particular position in the trace.

Given the Σ-algebra introduced before, let p be an assertion, c a constant in Σ, δ a Σ-term, and f and g a unary and binary function in Σ respectively. Then if ψ_1 and ψ_2 are statistical experiments, so are the following:

- *State Expression.* $p : \delta$, denoting the value of δ in the given position, if p holds at that position, otherwise the experiment is a failure.
- *Conjunction.* $\psi_1 \wedge_g \psi_2$, giving the value of g applied to the outcomes of ψ_1 and ψ_2, provided both ψ_1 and ψ_2 succeeded, otherwise the experiment is a failure.
- *Disjunction.* $\psi_1 \vee_g \psi_2$, same as above, except that only one of ψ_1 and ψ_2 has to succeed for the experiment to succeed.
- *Negation.* $\neg_c \psi_1$, denoting the value c if ψ_1 fails, and considered a failure otherwise.

- *Next.* $\bigcirc_f \psi_1$, denoting the value of f applied to the result of ψ_1 performed at the next position in the trace, provided ψ_1 succeeded.
- *Until.* $\psi_1 \mathcal{U}_g \psi_2$, denoting the value of g applied to the result of ψ_1 in the same position and the result of ψ_2 in the first position in which it succeeds, provided ψ_1 succeeded in all positions up to that point.

More formally, the *outcome value* of a statistical experiment ψ over a trace $\sigma : s_0, s_1, \ldots, s_n$ at a position $j \geq 0$ is written $[\psi]_{(\sigma,j)}$, and defined inductively as follows:

For a state expression:

$$[p : \delta]_{(\sigma,j)} = \begin{cases} s_j(\delta) & \text{if } s_j \vDash p \\ \bot & \text{otherwise} \end{cases}$$

where $s_j(\delta)$ is the value of δ in s.

For the boolean connectives:

$$[\neg_c \psi_1]_{(\sigma,j)} = \begin{cases} c & \text{if } [\psi_1]_{(\sigma,j)} = \bot \\ \bot & \text{otherwise} \end{cases}$$

$$[\psi_1 \wedge_g \psi_2]_{(\sigma,j)} = \begin{cases} g([\psi_1]_{(\sigma,j)}, [\psi_2]_{(\sigma,j)}) & \text{if } [\psi_1]_{(\sigma,j)} \neq \bot \text{ and } [\psi_2]_{(\sigma,j)} \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$[\psi_1 \vee_g \psi_2]_{(\sigma,j)} = \begin{cases} g([\psi_1]_{(\sigma,j)}, [\psi_2]_{(\sigma,j)}) & \text{if } [\psi_1]_{(\sigma,j)} \neq \bot \text{ or } [\psi_2]_{(\sigma,j)} \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

For the temporal operators:

$$[\bigcirc_f \psi_1]_{(\sigma,j)} = \begin{cases} f([\psi_1]_{(\sigma,j+1)}) & \text{if } [\psi_1]_{(\sigma,j+1)} \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$[\psi_1 \mathcal{U}_g \psi_2]_{(\sigma,j)} = \begin{cases} g([\psi_1]_{(\sigma,j)}, [\psi_2]_{(\sigma,k)}) & \text{where } k \text{ is the least } k, j \leq k \leq n, \\ & \text{such that } [\psi_2]_{(\sigma,k)} \neq \bot \text{ and} \\ & [\psi_1]_{(\sigma,i)} \neq \bot \text{ for every } i,\ j \leq i < k, \text{ or} \\ \bot & \text{if no such } k \text{ exists.} \end{cases}$$

In the case of disjunction and the case of the Until operator, one of the arguments of g may be $\perp$. We assume that g is extended to be defined in this case.

Example 1. Consider the trace

$$\sigma : \langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle, \langle 5,3\rangle, \langle 4,3\rangle$$

where each state $\langle x, y\rangle$ identifies the values of two integer variables x and y.

The following are simple examples of statistical experiments:

- The value of x in the first state of σ if $x < y$ holds, expressed by

  $$[x < y : x]_{(\sigma,0)}$$

  has outcome value $\perp$, since the first state does not satisfy $x < y$.
- The tuple $\langle x, y\rangle$ in the first state of σ if $x \leq y$, expressed by

  $$[x \leq y : \langle x, y\rangle]_{(\sigma,0)}$$

  has outcome value $\langle 1,1\rangle$.
- The experiment

  $$[\ (x \leq y : x)\ \ \mathcal{U}_{+}\ \ (y = x + 2 : y)\ ]_{(\sigma,0)}$$

  returns 4, the sum of the value of x in the first state and the value of y in the third state, as $\langle 1,3\rangle$ is the first state such that $y = x + 2$ is true.

Remark Note that linear-time temporal formulas with their usual interpretation over traces are a special case of statistical experiments where δ has sort $\{\mathrm{T}, \perp\}$ for all state expressions, and the operators have the following associated functions:

$$\wedge_{\wedge},\ \ \vee_{\vee},\ \ \neg_{T},\ \ \underset{id}{\bigcirc},\ \ \mathcal{U}_{\pi_2}$$

where id is the identity function and π_2 is projection onto the second element.

Thus a linear-time temporal formula can be considered a statistical experiment with value T if it holds and value $\perp$ if it fails, and can therefore also be used as subformulas in more complex queries. In the remainder of the paper we will use the usual notation for temporal formulas and omit the term δ and the functions associated with the operators.

2.3 Aggregate Statistics

Statistical experiments provide a value for a particular position in the trace. The outcomes of these experiments can be combined into *aggregate statistics* to obtain a value for part of the trace or the full trace. Examples of such

aggregate statistics are computing the minimum or maximum value of all successful experiments on a trace, or the sum of all outcomes, or just a count of all successfull experiments. We assume that these aggregate statistics can be computed in an incremental fashion and that the evaluation order, forward or backward, does not affect the final value. In addition, we assume that all aggregate statistics return $\perp$ if and only if all experiments fail.

An *aggregate expression* is defined as follows. Let φ be a statistical experiment, ψ a statistical experiment with sort boolean, g a binary function, and α an incrementally computable aggregate function. If ψ_1 and ψ_2 are aggregate expressions, so are the following:

- *Experiment:* φ. The value of the aggregate expression is equal to the outcome of the experiment at a particular position.
- *Conjunction:* $\psi_1 \wedge_g \psi_2$. The values of the aggregate expressions ψ_1 and ψ_2 are combined as described before for statistical experiments.
- *Disjunction:* $\psi_1 \vee_g \psi_2$. The values are combined as described above.
- *Unconditional Collection:* $\mathcal{C}_\alpha \psi_1$. The aggregate statistic α is applied to all outcomes of ψ_1 that are not $\perp$ from the current position until the end of the trace.
- *Interval Collection:* $\psi_1 \mathcal{I}_\alpha \psi$. The aggregate statistic α is applied to all outcomes of ψ_1 that are not $\perp$ over the maximal interval starting at the current position and ending when ψ ceases to hold.

Before we present a formal semantics of the last two operators, we show some examples of incrementally computable statistics, that is, functions α over traces $\sigma : s_0, s_1, \ldots, s_n$, such that there exists a binary function f_α such that

$$\alpha(\sigma) = f_\alpha(\ldots(f_\alpha(f_\alpha(\perp, s_n), s_{n-1}), \ldots), s_0)$$

Following are the binary functions that compute the aggregate statistics *minimum*, *maximum*, *sum*, and *count*. In the case that both arguments are non-$\perp$:

$$\begin{aligned} f_{min}(x,y) &= \text{if } x < y \text{ then } x \text{ else } y; \\ f_{max}(x,y) &= \text{if } x < y \text{ then } y \text{ else } x; \\ f_{\sum}(x,y) &= x + y \\ f_{count}(x,y) &= x + 1 \end{aligned}$$

For the case that one of the arguments is $\perp$ the above functions return the non-$\perp$ argument, except for f_{count}, which returns 1 if the first argument is $\perp$, and the non-$\perp$ argument if the second argument is $\perp$.

Given an incrementally computable statistic α the outcome of an aggregate

expression at a position $j \geq 0$ in trace σ is defined as follows

$$[\mathcal{C}_\alpha\, \psi_1]_{(\sigma,j)} \quad = f_\alpha(\, [\mathcal{C}_\alpha\, \psi_1]_{(\sigma,j+1)}\ ,\ [\psi_1]_{(\sigma,j)}\ )$$

$$[\psi_1\, \mathcal{I}_\alpha\, \phi]_{(\sigma,j)} = \begin{cases} f_\alpha(\, [\psi_1\, \mathcal{I}_\alpha\, \phi]_{(\sigma,j+1)}\ ,\ [\psi_1]_{(\sigma,j)}\ ) & \text{if } (\sigma, j) \vDash \phi \\ \bot & \text{otherwise} \end{cases}$$

Sometimes the value of the non-$\bot$ argument of a subexpression does not matter, for example in the scope of a counting aggregate. In the following we will simply omit the terms and function symbols in this case and assume a default labeling with a constant T.

Example 2. To illustrate the aggregate statistics consider the trace

$$\sigma : \langle 1,1,2\rangle, \langle 1,2,2\rangle, \langle 1,3,1\rangle, \langle 2,3,1\rangle, \langle 5,3,1\rangle, \langle 4,3,2\rangle$$

where each triple $\langle x, y, z\rangle$ identifies the values of three integer variables x, y and z. Below we show how various questions about this trace can be expressed as aggregate expressions and what their outcome values are.

- What is the number of positions such that the value of x is the same as the value of y?

  $$[\mathcal{C}_{count}(x = y)]_{(\sigma,0)} \quad = \quad count(T, \bot, \bot, \bot, \bot, \bot) \quad = \quad 1$$

  Note that the type of the expression here is the default, T.
- What is the minimum value of $x + y$ in the trace?

  $$[\mathcal{C}_{min}(true : x + y)]_{(\sigma,0)} \quad = \quad min(2,3,4,5,8,7) \quad = \quad 2$$

- What is the minimum difference between the value of x in some position where $x < y$ and that in the nearest position where $x \geq y$?

  $$[\mathcal{C}_{min}((x < y : x)\, \mathcal{U}_{|-|}\, (x \geq y : x))]_{(\sigma,0)} \quad = \quad min(\bot, 4, 4, 3, 5, \bot) \quad = \quad 3$$

  where $|-|$ is the absolute difference between the two arguments; if one of the arguments is $\bot$ (as is the case here for the fifth element in the sequence), it is defined to be equal to the non-$\bot$ argument.
- What is the average value of $x + y$?
  Although this is not directly definable here, we can define the average of a quantity as the pair of its sum and its count, that is we define

  $$\mathcal{C}_{avg}\varphi \ = \ \mathcal{C}_\Sigma\varphi \wedge_{\langle\cdot,\cdot\rangle} \mathcal{C}_{count}\varphi \ .$$

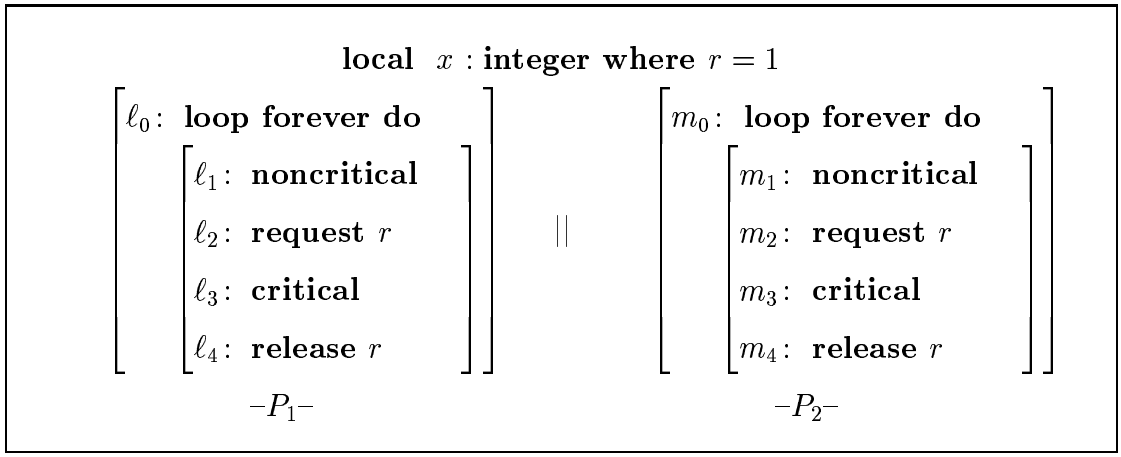

Fig. 1. Program MUX-SEM (mutual exclusion by semaphores)

Then the average value of $x + y$ can be written as

$$[\, \mathcal{C}_{avg}(true : x + y)\,]_{(\sigma,0)} \quad = \quad \langle 29, 6 \rangle$$

- What is the maximum number of times that $z = 1$ inside an interval where $x \leq y$?

$$[\, \mathcal{C}_{max}(\ (z = 1)\ \mathcal{I}_{count}\ (x \leq y)\ )\,]_{(\sigma,0)} \quad = \quad max(2, 2, 2, 1, \perp, \perp) \quad = \quad 2$$

In the next section we illustrate our specification language with two examples: a mutual exclusion algorithm and a communication protocol.

3 Examples

To illustrate the collection of statistics of running programs, we present two well known programs and some examples of relevant statistics for these programs. The programs are written in the Simple Programming Language (SPL) of [15], which is a Pascal-like language with constructs for concurrency. Statements are labeled to allow explicit reference to control locations.

3.1 Mutual Exclusion

Figure 1 shows a simple SPL program that ensures mutually exclusive access to the critical section of two processes by means of a semaphore[15]. The **request** statement is enabled only if r is positive, and when executed, it decrements r by 1. The **release** statement increments r by 1.

Following are some examples of statistics that can be monitored during program execution.

- *Semaphore values:* In a correct implementation, the maximum value of r should not exceed 1. The expression

$$\mathcal{C}_{max}\ (true : r)$$

  can be used to monitor whether this is indeed the case.

- *Mutual Exclusion:* The expression

$$\mathcal{C}_{max}\ (at\_\ell_3 : 1\ \vee_+\ at\_m_3 : 1)$$

  records the maximum number of processes present in the critical section at any one time. The predicate at_ℓ_3 is $true$ when process P_1 is in location ℓ_3; similarly, at_m_3 is $true$ when process P_2 is in location m_3. If the value of this expression exceeds 1, mutual exclusion is violated.

- *Bias:* The expression

$$\mathcal{C}_{count}(at\_\ell_3 \wedge \bigcirc \neg at\_\ell_3)\ \wedge_\div\ \mathcal{C}_{count}(at\_m_3 \wedge \bigcirc \neg at\_m_3)$$

  returns the ratio of the number of visits by P_1 to the critical section to the number of visits of P_2 to the critical section.

- *Overtaking:* Program MUX-SEM does not put a bound on how often one process can enter the critical section while the other process is waiting to enter. In practice, one may want to monitor the number of times a process is overtaken. The expression

$$\mathcal{C}_{max}\ ((\neg at\_m_3 \wedge\ \bigcirc(at\_m_3))\ \mathcal{I}_{count}\ (at\_\ell_2))$$

  records the maximum number of times P_2 visits the critical section during any period where P_1 idles at ℓ_2.

3.2 Communication Protocol

Figure 2 shows an SPL implementation (adapted from [16]) of the Alternating Bit Protocol, a communication protocol that guarantees data delivery to the receiver across a lossy channel, first proposed in [2]. Two processes, a *sender* and a *receiver* execute in parallel. The sender sends data items via the asynchronous data channel *dchan*; each data item is accompanied by a boolean value *seq* (the alternating bit). It then waits for the receiver to send an acknowledgement, consisting of one bit, on the asynchronous acknowledgement channel *achan*, or it times out (we assume that statement ℓ_4 is taken a fixed amount of time after it becomes enabled). If an ack was received and its value is equal to the seq bit, the sender assumes the data was received and it moves on to the next data item, simultaneously flipping the value of seq. If no ack was received, or its value was not equal to seq, the same data item is sent

again. The receiver retrieves the data items from *dchan*. If the accompanying seq bit is equal to its local ack bit, it accepts the data by moving its pointer to the next data item, and flips its ack bit. We assume that both *achan* and *dchan* may lose items, but do not corrupt or reorder items.

Following are some example queries on traces of this protocol.

- *Throughput:* The total number of data items successfully sent, can be expressed by
$$\mathcal{C}_{count}(\ at\_\ell_6 \wedge \bigcirc \neg at\_\ell_6)$$
- *Sent vs Received:* The number of items sent by the Sender versus the number of items received by the Receiver is recorded by
$$\mathcal{C}_{count}(at\_\ell_1 \wedge \bigcirc \neg at\_\ell_1) \wedge_{\langle .,. \rangle} \mathcal{C}_{count}(at\_m_1 \wedge \bigcirc \neg at\_m_1)$$
- *Maximum Retransmissions:* The maximum number of retransmissions for any one packet is expressed by
$$\mathcal{C}_{max}(\ (at\_\ell_1 \wedge \bigcirc(\neg at\_\ell_1))\ \mathcal{I}_{count}\ (\neg at\_\ell_6)\ )$$
The expression counts the number of times statement ℓ_1 is executed in any interval in which control does not reside at control location ℓ_6, where the current data item is updated. It then takes the maximum over all intervals.
- *Average Retransmissions:* The average number of retransmissions per packet can be expressed by a similar expression
$$\mathcal{C}_{avg}\left(\begin{array}{c}(at\_\ell_6 \wedge \bigcirc \neg at\_\ell_6) \\ \wedge_{\pi_2} \\ true\ \mathcal{U}_{\pi_2}\ ((at\_\ell_1\ \wedge\ \bigcirc(\neg at\_\ell_1))\ \mathcal{I}_{count}\ (\neg at\_\ell_6))\end{array}\right)$$
In each position the *Until* expression evaluates to the number of transmissions performed in the nearest interval where control is not at ℓ_6. The conjunction with $at\_\ell_6 \wedge \bigcirc \neg at\_\ell_6$ ensures that we count each interval only once in computing the average, as the sequence will have a non-$\perp$ value only in the positions where the sender moves to a new data item.

4 Evaluating Statistics

We now turn our attention to the problem of *evaluating* formulas for a given trace. Similar to the trace checking methods of [10], we use alternating automata as an intermediate representation. We define *algebraic* automata which produce a *value* when evaluated over traces. Our construction then consists of two steps: we first translate the formula into an equivalent automaton; then we traverse the automaton for the given trace to compute the result. The

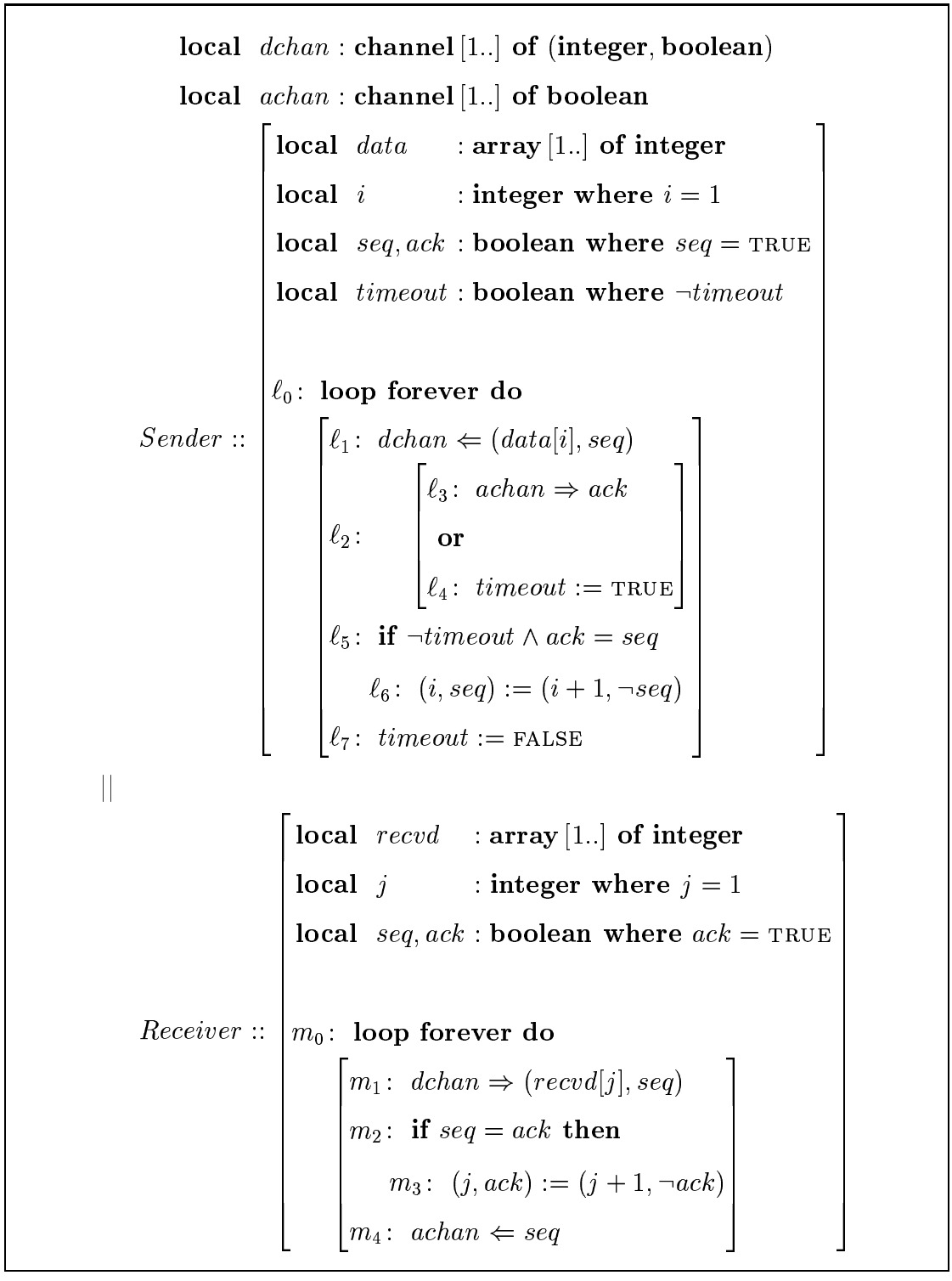

Fig. 2. Program ABP: Alternating bit protocol

motivation for this approach is to decouple the evaluation strategy from the definition of the temporal operators.

We begin with some basic definitions.

4.1 Algebraic Alternating Automata

Definition 1. (Algebraic Alternating Automaton). Let Σ be a many-sorted algebraic signature and X be a set of *program* variables. Let $g(x_1 : \tau_1, x_2 : \tau_2) : \tau$ be any term of sort τ over two variables $x_1 : \tau_1, x_2 : \tau_2 \notin X$, and $f(x_1)$ be a term of sort τ over one variable x_1 of sort τ_1. An *algebraic alternating automaton* (AAA) of sort τ is defined as follows:

$\mathcal{A} : \tau ::= \langle p, e \rangle$		terminal node
		with assertion p and $e : \tau \in \mathcal{T}(\Sigma, X)$
$\mid$	$\langle \mathcal{A} : \tau_1, f \rangle$	transient node
$\mid$	$\mathcal{A} : \tau_1 \wedge_g \mathcal{A} : \tau_2$	conjunction
$\mid$	$\mathcal{A} : \tau_1 \vee_g \mathcal{A} : \tau_2$	disjunction
$\mid$	$f(\mathcal{A} : \tau_1)$	function application

Note that the sort of any conjunction or disjunction operation of two automata is the sort of the term $g(x_1, x_2)$ that annotates the operation.

Example 3. Figure 3 shows an example of an AAA over the signature Σ, containing the single sort *nat*, a single constant $\perp$ for the undefined value, and the functions π_1, π_2, f_{min} and id. Nodes without outgoing edges denote terminal nodes; nodes with an outgoing edge are transient nodes $\langle \mathcal{A}, f \rangle$, with the edge leading to $\mathcal{A}$. The diamonds accompanied by an arc denote conjunction of the two branches and those without an arc denote disjunction.

Definition 2. (Value). Given a trace $\sigma : s_0, \ldots, s_{n-1}$ of length n and a position $i < n$, the *value* of an AAA $\mathcal{A}$ is defined by the function $eval(\mathcal{A}, \sigma, i)$,

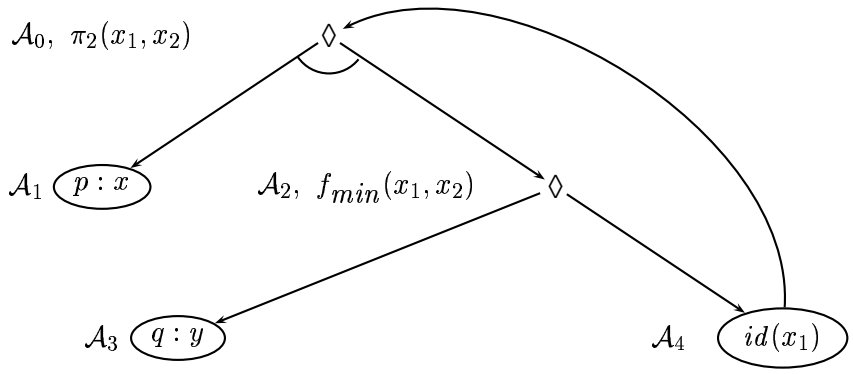

Fig. 3. Algebraic Alternating Automaton

given as follows

$$eval(\langle p, e\rangle, \sigma, i) = \begin{cases} [\sigma(i)](e) & \text{if } (\sigma, i) \vDash p \\ \bot & \text{otherwise} \end{cases}$$

$$eval(\langle \mathcal{A}_1, f\rangle, \sigma, i) = \begin{cases} f(eval(\mathcal{A}_1, \sigma, i+1)) & \text{if } eval(\mathcal{A}_1, \sigma, i+1) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$eval(\mathcal{A}_1 \wedge_g \mathcal{A}_2, \sigma, i) = \begin{cases} g(eval(\mathcal{A}_1, \sigma, i), eval(\mathcal{A}_2, \sigma, i)) \\ \quad \text{if } eval(\mathcal{A}_1, \sigma, i) \neq \bot \text{ and } eval(\mathcal{A}_2, \sigma, i) \neq \bot \\ \bot \quad \text{otherwise} \end{cases}$$

$$eval(\mathcal{A}_1 \vee_g \mathcal{A}_2, \sigma, i) = \begin{cases} g(eval(\mathcal{A}_1, \sigma, i), eval(\mathcal{A}_2, \sigma, i)) \\ \quad \text{if } eval(\mathcal{A}_1, \sigma, i) \neq \bot \text{ or } eval(\mathcal{A}_2, \sigma, i) \neq \bot \\ \bot \quad \text{otherwise} \end{cases}$$

$$eval(f(\mathcal{A}_1), \sigma, i) = f(eval(\mathcal{A}_1, \sigma, i))$$

The value of any automaton is defined to be $\bot$ at any position outside the trace, that is, for a trace of length n, any position $i \geq n$ has value $\bot$. Also note that for function application, f may have the ability to convert a non-$\bot$ value to $\bot$ and vice-versa.

Example 4. Consider again the automaton $\mathcal{A}_0$ in Figure 3 and let $\mathcal{V}$ be the Σ-

Positions	0	1	2	3	4	5	6
Assertion p	$\surd$	$\surd$	$\surd$				$\surd$
Assertion q	$\surd$	$\surd$		$\surd$		$\surd$	$\surd$
Program State	$\langle 3,1\rangle$	$\langle 1,2\rangle$	$\langle 2,1\rangle$	$\langle 2,3\rangle$	$\langle 0,3\rangle$	$\langle 6,2\rangle$	$\langle 1,0\rangle$

Fig. 4. Example program trace and satisfaction of assertions p and q

Algebra with carrier set $A = \mathcal{N} \cup \{\perp\}$, where $\mathcal{N}$ is the set of natural numbers, functions $\pi_1(x_1, x_2) = x_1$, $\pi_2(x_1, x_2) = x_2$, f_{min} the minimum function over integers that gives $\perp$ if both the arguments are $\perp$ and the non-$\perp$ value if one of the arguments is $\perp$, and id the identity function. We evaluate $\mathcal{A}_0$ over the trace tr shown in Figure 4. The trace shows the values of two variables x and y in each position and the satisfaction (indicated by $\surd$) of two assertions p and q over the program variables. The evaluation of $\mathcal{A}_0$ over tr is computed as follows

$$eval(\mathcal{A}_0, tr, i) = \begin{cases} eval(\mathcal{A}_2, tr, i) & \text{if } eval(\mathcal{A}_1, tr, i) \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

$$eval(\mathcal{A}_1, tr, i) = \begin{cases} tr(i)(x) & \text{if } tr(i) \models p \\ \perp & \text{otherwise} \end{cases}$$

$$eval(\mathcal{A}_2, tr, i) = \begin{cases} f_{min}(eval(\mathcal{A}_3, tr, i), eval(\mathcal{A}_4, tr, i)) \\ \quad \text{if } eval(\mathcal{A}_3, tr, i) \neq \perp \text{ or } eval(\mathcal{A}_4, tr, i) \neq \perp \\ \perp \quad \text{otherwise} \end{cases}$$

$$eval(\mathcal{A}_3, tr, i) = \begin{cases} tr(i)(y) & \text{if } tr(i) \models q \\ \perp & \text{otherwise} \end{cases}$$

$$eval(\mathcal{A}_4, tr, i) = id(eval(\mathcal{A}_0, tr, i+1)) = eval(\mathcal{A}_0, tr, i+1)$$

The outcome values of $eval$ are shown in Figure 5. Notice that the automaton $\mathcal{A}_0$ corresponds to the formula $\varphi_0 : \ q : y \ \mathcal{I}_{min} \ p$. As expected, the outcome values at the positions where p or q is false are $\perp$, and the outcome

Position	0	1	2	3	4	5	6	7
$\mathcal{A}_0$	1	2	$\perp$	$\perp$	$\perp$	$\perp$	0	$\perp$
$\mathcal{A}_1$	3	1	2	$\perp$	$\perp$	$\perp$	1	$\perp$
$\mathcal{A}_2$	1	2	$\perp$	3	$\perp$	0	0	$\perp$
$\mathcal{A}_3$	1	2	$\perp$	3	$\perp$	2	0	$\perp$
$\mathcal{A}_4$	2	$\perp$	$\perp$	$\perp$	$\perp$	0	$\perp$	$\perp$

Fig. 5. The result of *eval*

values at positions 0, 1, and 6 are the mimimum values of y over the interval where p is true.

4.2 Evaluation on Traces

Evaluation can proceed in the forward direction or in the reverse direction. The former strategy traverses the trace from the beginning to the end. The automaton is evaluated recursively, as dictated by the equations in Definition 2. Unfortunately, the complexity of forward evaluation is exponential in the length of the trace. This can be avoided, as pointed out by Rosu and Havelund [17], by traversing the trace backwards. We start by assigning the value $\perp$ to all nodes of the form $\langle \mathcal{A}, f \rangle$. The value of an automaton at a position $i < n$ depends only on the value of its sub-automata at the same position or, in case of a node of the form $\langle \mathcal{A}, f \rangle$, on the value of $\mathcal{A}$ in the next position of the trace (if any). Therefore, it is possible to perform a backwards evaluation while storing the values of all automata at the current and the next positions only.

5 Translating Specifications to Automata

In this section, we describe the translation of formulas of the query language to the algebraic alternating automata described in the previous section.

We assume that the algebraic signature Σ contains a binary function f_α for the incremental computation of each aggregate statistic α. In addition, to model the negation operator, we assume that for each constant c in Σ there exists a function $negate_c$ defined as

$$negate_c(v) = \begin{cases} \perp & \text{if } v \neq \perp \\ c & \text{otherwise} \end{cases} .$$

We also assume that each sort has the identity function id.

Given a statistical experiment ψ, its corresponding AAA $\mathcal{A}_A(\psi)$ is constructed as follows.

For a state expression and the boolean connectives:

$$\begin{aligned}
\mathcal{A}_A(\psi : e_j) &= \langle \psi, e_j \rangle \\
\mathcal{A}_A(\psi_1 \ \wedge_g \ \psi_2) &= \mathcal{A}_A(\psi_1) \ \wedge_g \ \mathcal{A}_A(\psi_2) \\
\mathcal{A}_A(\psi_1 \ \vee_g \ \psi_2) &= \mathcal{A}_A(\psi_1) \ \vee_g \ \mathcal{A}_A(\psi_2) \\
\mathcal{A}_A(\neg_c \psi_1) &= negate_c(\mathcal{A}_A(\psi_1))
\end{aligned}$$

For the temporal operators:

$$\begin{aligned}
\mathcal{A}_A(\bigcirc_f(\psi)) &= \langle \mathcal{A}_A(\psi), f \rangle \\
\mathcal{A}_A(\psi_1 \mathcal{U} \psi_2) &= f(\ \mathcal{A}_1\ )
\end{aligned}$$

with

$$\mathcal{A}_1 \ = \ (\mathcal{A}_A(\psi_1) \wedge_{\langle x_1, \pi_2(x_2) \rangle} \langle \mathcal{A}_1, id \rangle) \ \vee_{collect} \ \mathcal{A}_A(\psi_2)$$

where the function *collect* is defined by

$$collect(x_1, x_2) = \begin{cases} \langle \pi_1(x_1), x_2 \rangle & if \ x_2 \neq \perp \\ x_1 & \text{otherwise} \end{cases}$$

The first argument x_1 is a tuple containing the value of ψ_1 in the current state and the value of ψ_2 from the ψ_2 state nearest to the next position in the trace. The second argument x_2 represents the value of ψ_2 in the current state. If there is a non-$\perp$ ψ_2 value in the current state, the old value of ψ_2 is discarded and the current value is chosen.

The construction of the $\mathcal{U}$ automaton is illustrated in Figure 6. Automaton $\mathcal{A}_2$ computes a tuple consisting of the current value of ψ_1 and the value of ψ_2 nearest to the next position in the trace. The node $\mathcal{A}_0$ computes the same tuple as $\mathcal{A}_1$ except that it applies the function f to the tuple as indicated in the figure.

Given an aggregate expression ψ the corresponding AAA $\mathcal{A}_A(\psi)$ is constructed as above for conjunction and disjunction. The constructions for the unconditional and interval collection are as follows:

$$\begin{aligned}
\mathcal{A}_A(\mathcal{C}_\alpha(\psi)) &= \mathcal{A}_A(\psi) \ \vee_{f_\alpha} \ \langle \mathcal{A}_A(\mathcal{C}_\alpha(\psi)), id \rangle \\
\mathcal{A}_A(\psi \mathcal{I}_\alpha \varphi) &= (\langle \ \mathcal{A}_A(\psi \mathcal{I}_\alpha \varphi), id \rangle \vee_{f_\alpha} \mathcal{A}_A(\psi)) \ \wedge_{\pi_1} \ \mathcal{A}_A \varphi
\end{aligned}$$

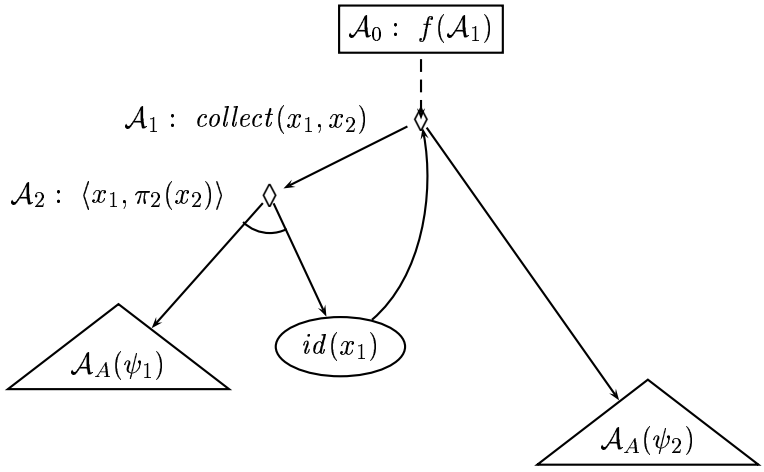

Fig. 6. Automata construction for $\psi_1 \mathcal{U} \psi_2$

The constructions of these automata are shown in Figure 7. In the construction of the $\mathcal{C}_\alpha$ operator, the node labelled with the identity function $id(x_1)$ collects the value of the statistic in the next state of the trace (which is initialized to $\perp$ at the end of the trace).

Example 5. Figure 8 shows the AAA for the formula

$$\mathcal{C}_{avg}(\ at\_\ell_6 \wedge_{\pi_2} \ \bigcirc((at\_\ell_1 \ \wedge \ \bigcirc \neg at\_\ell_1)\ \mathcal{I}_{count}\ (\neg at\_\ell_6))\ )\ ,$$

the formula for calculating the number of retransmissions in the alternating bit protocol example from Section 3.2.

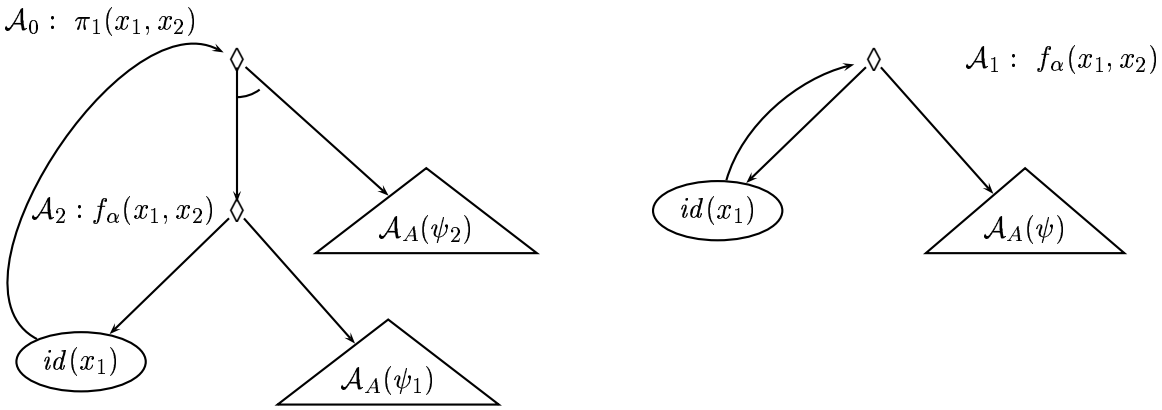

Fig. 7. Automata construction for the operators $\psi_1 \mathcal{I}_\alpha \psi_2$ (left) and $\mathcal{C}_\alpha \psi$ (right).

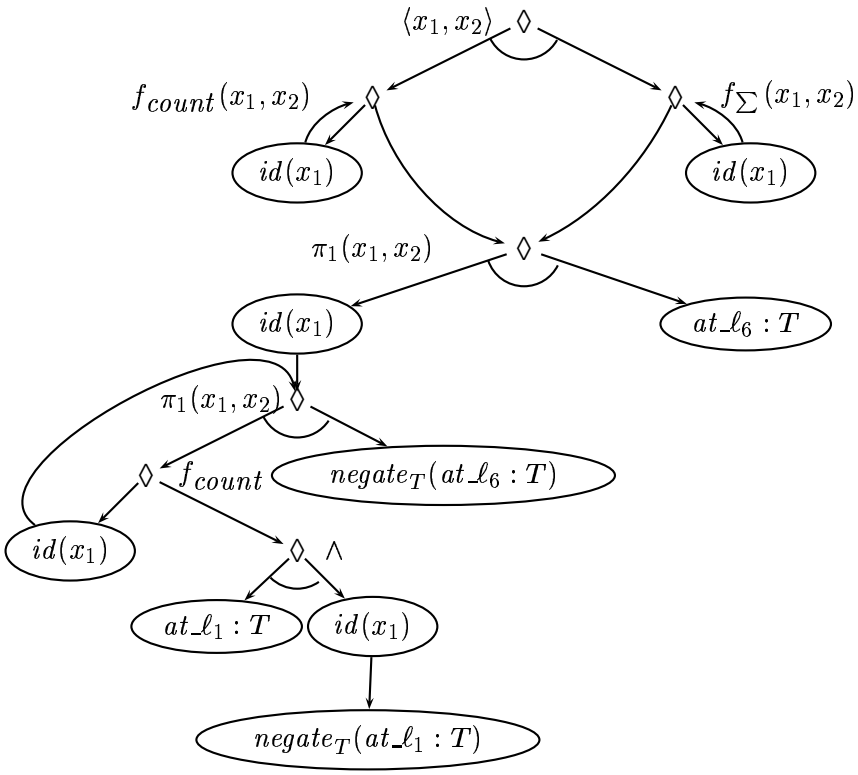

Fig. 8. Translation for $\mathcal{C}_{avg}[at\_\ell_6 \wedge_{\pi_2} \bigcirc((at\_\ell_1 \wedge \bigcirc \neg at\_\ell_1)\, \mathcal{I}_{count}\, (\neg at\_\ell_6))]$

6 Experimental Results

The evaluation algorithm from Section 4.2 has been implemented in Java, making use of existing software modules for expression parsing and propositional simplification available in the STeP (Stanford Temporal Prover) system [3]. The formulas described in the mutual exclusion and alternating bit protocol examples of Section 3.1 along with some other formulas on these examples were hand translated following the translation described in 5. Traces of varying length were generated by simulating the SPL programs. At each position a single step of a randomly chosen process was executed. We then measured the time taken for evaluation by means of backward evaluation over traces of varying length. The results are shown in Figure 9. The times were measured for a 1.7GHz PC, running Redhat Linux v7.0 and Sun JDK 1.4.

Acknowledgements

We would like to thank the anonymous referees for their thorough reading of our submission and their many constructive comments and suggestions.

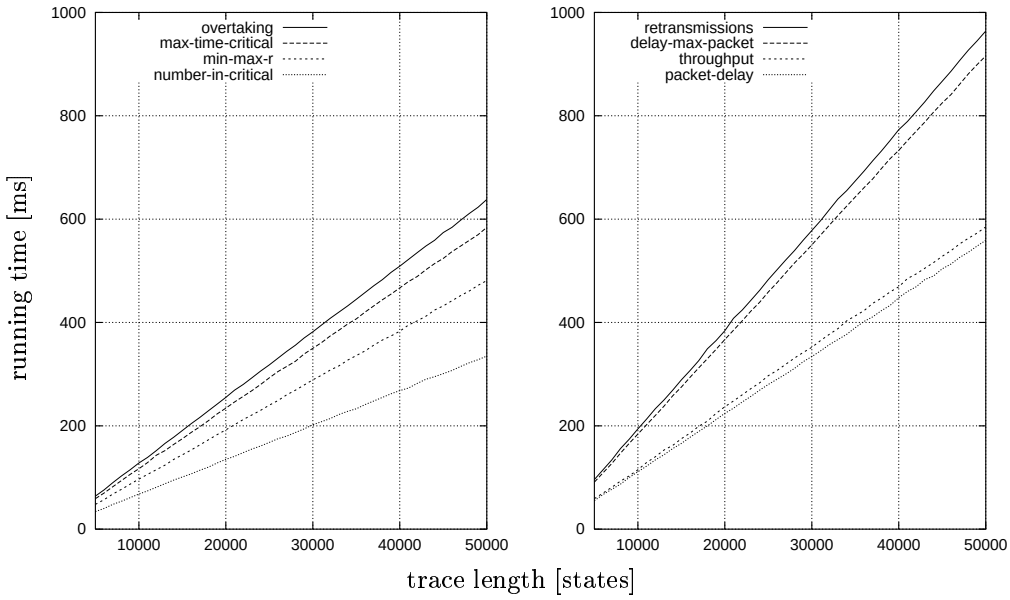

Fig. 9. Running times for queries on the Mutual Exclusion program (left) and the Alternating Bit Protocol (right).

References

[1] Alur, R., S. L. Torre, K. Ettessami and D. Peled, *Parametric temporal logic for model measuring*, in: J. Wiedermann, P. van Emde Boas and M. Nielsen, editors, *ICALP'99, Prague, Czech Republic*, LNCS **1644** (1999), pp. 159–168.

[2] Bartlett, K., R. Scantlebury and P. Wilkinson, *A note on reliable full-duplex transmission over half-duplex links*, Communications of the ACM **12** (1969), pp. 260–261.

[3] Bjørner, N. S., A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H. B. Sipma and T. E. Uribe, *Verifying temporal properties of reactive systems: A STeP tutorial*, Formal Methods in System Design **16** (2000), pp. 227–270.

[4] Bruns, G. and P. Godefroid, *Temporal logic query checking*, in: *Proc. 16th IEEE Symp. Logic in Comp. Sci.* (2001), pp. 409–417.

[5] Chakrabarti, P., P. Dasgupta, J. Deka and S. Sankaranarayanan, *Min-max computation tree logic*, Artificial Intelligence **127** (2001), pp. 137–162.

[6] Chandra, A. K., D. C. Kozen and L. J. Stockmeyer, *Alternation*, J. ACM **28** (1981), pp. 114–133.

[7] Drusinsky, D., *The Temporal Rover and the ATG Rover*, in: K. Havelund, J. Penix and W. Visser, editors, *SPIN Model Checking and Software Verification, 7^{t}h Int'l SPIN Workshop*, LNCS **1885** (2000), pp. 323–330.

[8] Emerson, A., A. Mok, A. P. Sistla and J. Srinivasan, *Quantitative temporal reasoning*, Real Time Systems **4** (1993), pp. 334–351.

[9] Emerson, A. and R. Trefler, *Generalized quantitative temporal reasoning: An automata-theoretic approach*, in: *TAPSOFT: 7th International Joint Conference on Theory and Practice of Software Development*, 1997.

[10] Finkbeiner, B. and H. Sipma, *Checking finite traces using alternating automata*, in: K. Havelund and G. Rosu, editors, *Electronic Notes in Theoretical Computer Science*, Electronic Notes in Theoretical Computer Science **55** (2001), pp. 1–17.

[11] Graham, S. L., P. B. Kessler and M. K. McKusick, *gprof: a call graph execution profiler*, in: *SIGPLAN Symposium on Compiler Construction*, 1982, pp. 120–126.

[12] Havelund, K., *Using runtime analysis to guide model checking of java programs*, in: K. Havelund, J. Penix and W. Visser, editors, *SPIN Model Checking and Software Verification, 7^{t}h Int'l SPIN Workshop*, LNCS **1885** (2000), pp. 245–264.

[13] Havelund, K. and G. Rosu, editors, "Runtime Verification 2001," Electronic Notes in Theoretical Computer Science **55**, Elsevier Science Publishers, 2001.

[14] Manna, Z. and A. Pnueli, *Specification and verification of concurrent programs by $\forall$-automata*, in: B. Banieqbal, H. Barringer and A. Pnueli, editors, *Temporal Logic in Specification*, number 398 in LNCS, Springer-Verlag, Berlin, 1987 pp. 124–164, also in *Proc. 14th ACM Symp. Princ. of Prog. Lang.*, Munich, Germany, pp. 1–12, January 1987.

[15] Manna, Z. and A. Pnueli, "Temporal Verification of Reactive Systems: Safety," Springer-Verlag, New York, 1995.

[16] Manna, Z. and A. Pnueli, "Temporal Verification of Reactive Systems: Progress," Springer-Verlag, New York, 1996, draft manuscript.

[17] Rosu, G. and K. Havelund, *Synthesizing dynamic programming algorithms from linear temporal logic formulae*, 2001, submitted for publication.

[18] Vardi, M. Y., *Alternating automata and program verification*, in: J. van Leeuwen, editor, *Computer Science Today. Recent Trends and Developments*, LNCS **1000**, Springer-Verlag, 1995 pp. 471–485.

[19] Vardi, M. Y., *An automata-theoretic approach to linear temporal logic*, in: F. Moller and G. Birtwistle, editors, *Logics for Concurrency. Structure versus Automata*, LNCS **1043** (1996), pp. 238–266.