 Electronic Notes in Theoretical Computer Science 99 (2004) 245–266 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Automatic Analysis of the NRL Pump [*\**](#_bookmark0)

# Ruggero Lanotte[a](#_bookmark1), Andrea Maggiolo-Schettini[b](#_bookmark2), Simone Tini[a](#_bookmark1),

Angelo Troina[b](#_bookmark2) and Enrico Tronci[c](#_bookmark3)

a *Dipartimento di Scienze della Cultura, Politiche e dell’Informazione, Universit`a dell’Insubria,*

*Via Valleggio 11, 22100 Como, Italy*

b *Dipartimento di Informatica, Universita` di Pisa, Via Buonarroti 2, 56127 Pisa, Italy*

c *Dipartimento di Informatica, Universita` di Roma “La Sapienza”, Via Salaria 113, 00198 Roma, Italy*

Abstract

We define a probabilistic model for the NRL Pump and using FHP-mur*ϕ* show experimentally that there exists a *probabilistic* covert channel whose capacity depends on various NRL Pump parameters (e.g. buffer size, number of samples in the moving average, etc).

# Introduction

A computer system may store and process information with a range of clas- sification levels and provide services to users with a range of clearances. The system is said to be *multilevel secure* [[3](#_bookmark36)] if users have access to information classified at or below their clearance, and are prevented from accessing infor- mation classified above their clearance.

In a distributed framework, multilevel security can be achieved by using multilevel secure components to connect single-level systems at different secu- rity levels, thus creating a *multiple single-level security* architecture [[10](#_bookmark40),[11](#_bookmark41)].

The role of multilevel secure components is to minimize leaks of high level information from high level systems to lower level systems. A proposal for this

Research partially supported by progetto cofinanziato “Metodi Formali per la Sicurezza e il Tempo”(MEFISTO)

1571-0661 © 2004 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2004.02.011

component is the NRL Pump [[13](#_bookmark44),[14](#_bookmark45),[12](#_bookmark42)], which is a trusted device that acts as a router forwarding messages from a low-level system to a high-level system, by monitoring the timing of the acknowledgments in the opposite way. More precisely, the NRL Pump lets information be securely sent from a system at a lower security level to one at a higher security level. Now, if the high system passed acknowledgments directly to the low system, then, the high system, by altering the acknowledgment timing, could pass information to the low system. To avoid such *covert channel*, while maintaining acknowledgments to have a reliable communication, the pump decouples the acknowledgment stream. As, however, for performance reasons, the long-term high-system- to-pump behavior should be reflected in the long-term low-system-to-pump behavior, the pump uses statistically modulated acknowledgments.

In [[12](#_bookmark42)] a NRL Pump assurance strategy is proposed. A logical view of the pump is refined using a combination of Statemate activity charts and state charts. The activities and behavior of the logical design are mapped to an implementation described in terms of Statemate module charts. A Verdi specification expresses the requirements to which the implementation must be shown to conform. Verification is done through proof using the EVES verification system, and by testing using WhiteBoxDeepCover tool.

In this paper we model the NRL Pump by Probabilistic Timed Automata [[2](#_bookmark34),[15](#_bookmark46),[1](#_bookmark35)]. These automata and requirements on their behavior are translated into specifications for the FHP-mur*ϕ* tool [[5](#_bookmark37),[6](#_bookmark38),[17](#_bookmark48)]. Using this tool, we show experimentally that there exists a probabilistic covert channel whose capacity depends on various NRL Pump parameters (e.g. buffer size, number of samples in the high-system-to-pump moving average, number of samples in the pump- to-low-system moving average, *.. .*).

In section [2](#_bookmark4) we describe the NRL Pump and its security problem. In section [3](#_bookmark6) we introduce Probabilistic Timed Automata. In section [4](#_bookmark8) we use the automata to model the NRL Pump. In section [5](#_bookmark12) we discuss the NRL Pump probabilistic ack delay modeling. In section [6](#_bookmark16) we give the FHP-Mur*ϕ* model of the NRL Pump. In section [7](#_bookmark31) we show our experimental results.

# The NRL Pump

The NRL Pump is a special purpose-device that forwards data from low level agents to high level agents, but not conversely. More precisely, the pump works as follows:

1. A low agent sends a message to some high agent through the pump.
2. The pump stores the message in a buffer and sends an acknowledgment to the low agent, in order to make the communication reliable. The time

of the acknowledgment is probabilistically based on a moving average of acknowledgment times from the high agents to the pump. In this way, the high agent cannot alter the acknowledgment timing to send infor- mation to the low agent. Moreover, the long-term high-agents-to-pump behavior is reflected by the long-term pump-to-low-agents behavior, so that performance is not degraded.

1. The low agent cannot send any new message until the acknowledgment of the previous message is received.
2. The pump stores the message until the high agent is able to receive it.
3. The high agent receives the message and, then, sends an acknowledgment to the pump.
4. If the high agent does not acknowledge some received message before a fixed timeout expires [1](#_bookmark5) , the pump stops the communication.

The validation protocol that models the initialization of the communication between the pump and the low and high systems is defined as follows (*LS* represents the low system, *P* represents the pump, *HS* represents the high system and *A* → *B* : *msg* represents the message *msg* sent from *A* to *B*):

*LS* → *P* : *reqL* the low system requests to the pump to start a communication with the high system

*P* → *LS* : *validL* the pump checks if the low system is a valid process, and, then, it acknowledges its request

*P* → *HS* : *reqH* the pump requests to the high system to start a communication with the low system

*HS* → *P* : *validH* the high system checks if the pump is a valid process, and, then, it acknowledges its request

*P* → *HS* : *grantH* the pump communicates to the high system that the communication can start

*P* → *LS* : *grantL* the pump communicates to the low system that the communication can start

1 Such a timeout is defined by the pump-administrator.

Data communication can be modeled as follows:

*LS* → *P* : *sendL* the low system sends to the pump data to deliver to the high system

*P* → *LS* : *ackL* the pump acknowledges to the low system with a random delay

*P* → *HS* : *sendH* the pump sends the data to the high system

*HS* → *P* : *ackH* the high system acknowledges to the pump

When the low system receives the ack related to the last message, it ends the communication by sending a closure message:

*LS* → *P* : *close*

When the pump does not receive any ack from the high system before the expiration of the timeout, it closes the connection by sending to the low system an exit message:

*P* → *LS* : *exit*

When the low system tries to restore a communication stopped by an exit message from the pump, the pump forgets all the messages in the buffer undelivered to the high system.

# Probabilistic Timed Automata

We give a definition of Probabilistic Timed Automata which differs slightly from those in [[2](#_bookmark34),[15](#_bookmark46),[1](#_bookmark35)].

Let us assume a set *X* of *integer variables*, with a subset *Y* of variables called *clocks*. A *valuation* over *X* is a mapping *v* : *X* → Z assigning natural values to clocks and integer values to variables in *X* \ *Y* . For a valuation *v* and a time value *t* ∈ N, let *v*+*t* denote the valuation such that (*v*+*t*)(*x*)= *v*(*x*), for each integer variable *x* ∈ *X* \*Y* , and (*v* + *t*)(*y*)= *v*(*y*)+ *t*, for each clock *y* ∈ *Y* .

The set of *constraints* over *X*, denoted Φ(*X*), is defined by the following grammar, where *φ* ranges over Φ(*X*), *x* ∈ *X*, *c* ∈ Z and ∼∈ {*<,* ≤*,* =*,* /=*, >,* ≥

}:

*φ* ::= *x* ∼ *c* | *φ* ∧ *φ* | ¬*φ* | *φ* ∨ *φ* | *true*

We write *v* |= *φ* when *the valuation v satisfies the constraint φ*. Formally,

*v* |= *x* ∼ *c* iff *v*(*x*) ∼ *c*, *v* |= *φ*1 ∧ *φ*2 iff *v* |= *φ*1 and *v* |= *φ*2, *v* |= ¬*φ* iff *v* |= *φ*,

*e*0 = (*q*0 *, b, true,* ∅*, q*1) *πq*0 *,b*(*e*0)= 1

*e*1 = (*q*0 *, a, true,* ∅*, q*2) *πq ,a*(*e*1 )= 1

0 3

*e*2 = (*q*0 *, a, true,* ∅*, q*3) *πq ,a*(*e*2 )= 2

0 3

*e*∈*start*(*q*0 *,a*) 0 *,,*

P *πq ,a*(*e*)= 1

P *π* (*e*)= 1 *a,* 2 *q*3

*e*∈*start*(*q*0 *,b*)

*q*0 *,b*

3 *J*

*,, ,*v*,* *,,*

3

*q* ( *b,* 1 *q*

1

0

*a,* 1

) *q*2

*Jx* ≤ 5 *J J*

Fig. 1. Example of Probabilistic Timed Automaton

*v* |= *φ*1 ∨ *φ*2 iff *v* |= *φ*1 or *v* |= *φ*2, and *v* |= *true*.

An *assignment* over *X* is a set of expressions either of the form *x*' = *c* or of the form *x*' = *y* + *c*, where *x, y* ∈ *X* and *c* ∈ Z.

With *Ass*(*X*) we denote the set of sets of assignments {*x*' = *e*1*,..., x*' =

1

*n*

*en*} such that *xi* ∈ *X* and *xi* /= *xj*, for any *i* /= *j*.

Let *B* ∈ *Ass*(*X*); with *v*[*B*] we denote the valuation resulting after the assignments in *B*. More precisely, *v*[*B*](*x*) = *c* if *x*' = *c* is in *B*, *v*[*B*](*x*) = *v*(*y*)+ *c* if *x*' = *y* + *c* is in *B*, and *v*[*B*](*x*)= *v*(*x*), otherwise.

Definition 3.1 *A* Probabilistic Timed Automaton *(PTA for short) is a 6- tuple A* = (Σ*, X, Q, q*0*, δ, π*)*, where:*

* Σ *is a finite alphabet of* actions*.*
* *X is a finite set of* variables *with a subset Y of* clocks*.*
* *Q is a finite set of* states *and q*0 ∈ *Q is the* initial state*.*
* *δ* ⊆ *Q* × Σ ∪ {*τ* } × Φ(*X*) × *Ass*(*X*) × *Q is a finite set of* transitions*. The symbol τ represents the silent or internal move. For a state q, we de- note with start*(*q*) *the set of transitions with q as source state, i.e. the set*

{(*q*1*, a, φ, B, q*2) ∈ *δ* | *q*1 = *q*}*; and we denote with start*(*q, a*) *the set of tran- sitions with q as source state and a as action, i.e. the set* {(*q*1*, α, φ, B, q*2) ∈ *δ* | *q*1 = *q and α* = *a*}*.*

* *π* = {*πq,a* : *start*(*q, a*) → (0*,* 1] | *q* ∈ *Q, a* ∈ Σ ∪ {*τ* }} *is a family of* prob- ability functions *πq,a such that πq,a*(*e*) *is the probability of performing the transition e in state q if action a is chosen. We require that, for each prob- ability function πq,a, it holds that e*∈*start*(*q,a*) *πq,a*(*e*) ∈ {0*,* 1}*. Intuitively, an automaton chooses non-deterministically the kind of action it wants to*

Σ

*take in a state, and then makes a probabilistic choice among the actions of that type it can perform in that state. See Figure* [*1*](#_bookmark7) *for an example.*

Let us explain now the behavior of a PTA A.

A *configuration* of *A* is a pair *s* = (*q, v*), where *q* ∈ *Q* is a state of *A*, and

*v* is a valuation over *X*.

The set of all the configurations of *A* is denoted with S*A*.

There is a *step* from a configuration *s*1 = (*q*1*, v*1) to a configuration *s*2 = (*q*2*, v*2) through action *a* ∈ Σ ∪ {*τ* }, after time *t* ∈ N, and with probability *p*,

(*a,t,p*)

written *s*1 −→ *s*2, if there is a transition *e* = (*q*1*, a, φ, B, q*2) ∈ *δ* such that

(*v*1 + *t*) |= *φ*, *πq*1*,a*(*e*)= *p >* 0 and *v*2 = (*v*1 + *t*)[*B*].

Given a configuration *s*, with *Adm*(*s*) we denote the set of pairs (*a, t*) such

that from *s* there is a step *s* (*a,t,p*) *s*' to some configuration *s*'.

−→

A configuration *s* = (*qi, vi*) is called *terminal* iff *Adm*(*s*) = ∅; we denote with *St* the set of the terminal configurations.

An *execution fragment* starting from *s*0 is a finite sequence of steps *σ* =

(*a*1 *,t*1 *,p*1)

*s*0 −→ *s*1

(*a*2 *,t*2 *,p*2)

−→ *s*2

(*a*3 *,t*3 *,p*3)

−→ *...*

(*ak,tk,pk*)

−→ *sk*. We define *last*(*σ*)= *sk*, |*σ*| = *k*,

and *step*(*σ, k*) = (*ak, tk*). For any *j < k*, with *σj* we define the sequence of

steps *s*

(*a*1 *,t*1 *,p*1)

*s*

*...*

(*aj,tj,pj* )

*s* .

0 −→ 1 −→ −→ *j*

If |*σ*| = 0 we put *P* (*σ*)= 1, else, if |*σ*| = *k* ≥ 1, we define *P* (*σ*)= *p*1 ·*.. .*·*pk*.

The execution fragment *σ* is called *maximal* iff *last*(*σ*) ∈ *St*. We denote with *ExecFrag*(*s*) the set of execution fragments starting from *s*.

An *execution* is either a maximal execution fragment or an infinite sequence

(*a*1 *,t*1 *,p*1)

*s*0 −→ *s*1

(*a*2 *,t*2 *,p*2)

−→ We denote with *Exec*(*s*) the set of executions starting

from *s*. Finally, let *σ* ↑ denote the set of executions *σ*' such that *σ* ≤*prefix σ*', where *prefix* is the usual prefix relation over sequences.

Executions and execution fragments of a PTA arise by resolving both the nondeterministic and the probabilistic choices [[15](#_bookmark46)]. We introduce now *sched- ulers* of PTAs as functions that resolve all the nondeterministic choices of the model.

A *scheduler* of a PTA *A* = (Σ*, X, Q, q*0*, δ, π*) is a function *F* mapping every execution fragment *σ* of *A* to a pair (*a, t*), where *a* ∈ Σ and *t* ∈ N, such that (*a, t*) ∈ *Adm*(*last*(*σ*)). With F we denote the set of all schedulers of *A*.

For a scheduler *F* of a PTA *A* we define *ExecFragF* as the set of execution fragments *σ* such that *step*(*σ, k*) = *F* (*σj*) for all 1 ≤ *j* ≤ |*σ*|, and *ExecF* as the set of executions *σ* such that *step*(*σ, k*)= *F* (*σj*) for all 1 ≤ *j* ≤ |*σ*| if *σ* is terminal, or for all *j* ∈ N if *σ* is infinite.

Assuming the basic notions of probability theory (see e.g. [[8](#_bookmark43)]) we define the probability space on the executions starting in a given configuration *s* ∈ S*A* as follows. Given a scheduler *F* , let *ExecF* (*s*) be the set of executions starting in *s*, *ExecFragF* (*s*) be the set of execution fragments starting in *s*, and Σ*F ield*(*s*) be the smallest sigma field on *ExecF* (*s*) that contains the basic cylinders *σ* ↑,

where *σ* ∈ *ExecFragF* (*s*).

The probability measure *Prob* is the unique measure on Σ*F ield*(*s*) such that

*Prob*(*σ* ↑)= *P* (*σ*).

* 1. *Parallel composition*

Given two PTAs *A*1 and *A*2 with the same integer variables *X*, we define the parallel composition of *A*1 and *A*2, denoted *A*1||*p A*2, where *p* ∈ [0*,* 1] and *L* ⊆ Σ. The set of states of *A*1||*p A*2 is given by the cartesian product of the states of the two automata *A*1 and *A*2. Intuitively, *A*1 and *A*2 synchronize on actions in *L*, and, if some action not in *L* is performed, then *A*1 computes with probability *p*, and *A*2 computes with probability 1 − *p*. Formally, given a state (*r, q*) of *A*1||*p A*2, the set of transitions starting from (*r, q*) is obtained by the following rules:

*L*

*L*

*L*

* + - If from state *r* the PTA *A*1 has a transition (*r, a, φ, B, r*') with action *a* /∈ L- and probability *p*', and *A*2 does not have in state *q* any transition with action *a*, then *A*1||*p A*2 has a transition ((*r, q*)*, a, φ, B,* (*r*'*, q*)) with probability *p*'.

*L*

* + - If from state *q* the PTA *A*2 has a transition (*q, a, φ, B, q*') with action *a* /∈ L- and probability *p*', and *A*1 does not have in state *r* any transition with action *a*, then *A*1||*p A*2 has a transition ((*r, q*)*, a, φ, B,* (*r, q*')) with probability *p*'.

*L*

* + - If from state *r* the PTA *A*1 has a transition (*r, a, φ, B, r*') with action *a* /∈ L- and probability *p*', and also *A*2 can perform in state *q* a transition with action *a*, then *A*1||*p A*2 has a transition ((*r, q*)*, a, φ, B,* (*r*'*, q*)) with probability *p*' · *p*.

*L*

* + - If from state *q* the PTA *A*2 has a transition (*q, a, φ, B, q*') with action *a* /∈ L- and probability *p*', and also *A*1 can perform in state *r* a transition with action *a*, then *A*1||*p A*2 has a transition ((*r, q*)*, a, φ, B,* (*r, q*')) with probability *p*' ·

*L*

(1 − *p*).

* + - If from state *r* the PTA *A*1 has a transition (*r, a, φ*1*, B*1*, r*') with action

*a* ∈ L-

and probability *p*', and from state *q* the PTA *A*2 has a transition

(*q, a, φ*2*, B*2*, q*') with probability *p*'' and *B*1 ∪ *B*2 ∈ *Ass*(*X*), then *A*1 and *A*2 synchronize and therefore *A*1||*p A*2 has a transition ((*r, q*)*, a, φ*1 ∧ *φ*2*, B*1 ∪ *B*2*,* (*r,*' *q*')) with probability *p*' · *p*''.

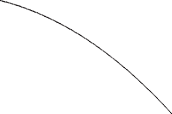
*L*

# Modeling the NRL Pump with PTAs

In this section we show how the NRL Pump presented in Section [2](#_bookmark4) can be modeled by PTAs. We model the pump as the parallel composition of the PTAs *Pump*1 and *Pump*2 depicted in Figure [4](#_bookmark11). The Low and High Systems are depicted in Figures [2](#_bookmark9) and [3](#_bookmark10). For simplicity, we have omitted constraints

*exit*

{*exit, close*}



)*,*ss*,req* )*,,valid*)*,,gran*)*t*

*,,send*)*,,*

*L L*

*q*1 *q*2 *q*3

*L L*

*q*4 *q*5

*J J*

*J J J*

¬

*ackL*

Fig. 2. *LS*: The Low System

*exit*

{*exit, close*}

)*,*ss*,req* )*,,valid*)*,,gran*)*t*

*,,send*)*,,*

*H H*

*r*1 *r*2 *r*3

*H H*

*r*4 *r*5

*J J*

*J J J*

¬

*ackH*

Fig. 3. *HS*: The High System

*true* and the probabilities when they are equal to 1.

The low system starts the communication by sending a request to the pump (*reqL*), waits for its validation (*validL*) and the start message (*grantL*). Now, in state *q*4, the low system starts sending its data to the pump (*sendL*) and receiving the related acknowledgment (*ackL*). If the communication ends in a correct way, it stops the communication by sending a *close* message to the pump, if the communication ends in an incorrect way (for instance if the high system does not acknowledge to the pump), the pump stops the connection by executing an *exit* action.

The high system enters the communication process by receiving a request from the pump (*reqH*), sending its validation message (*validH*) and waiting for the start message (*grantH*). Now, in state *r*4, the high system starts receiving data from the pump (*sendH*) and sending back the related acknowledgment (*ackH*). If the communication ends in a correct way, it receives the *close* message from the low system through the pump, if the communication ends in an incorrect way (for instance if it does not acknowledge to the pump before the timeout expires), the pump stops the connection by executing an *exit* action.

In states *g*1*,... , g*7 the automaton *Pump*1 initializes the connection be- tween the high and the low system.

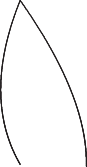
)*,,req*)*L ,,vali*)*dL,,reqH*)*,,*

*g*1

*g*2

*g*3

*g*4

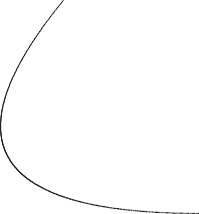
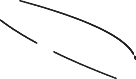
‘`*J J*

*J vaJlidH*

*,*v*,*

*exit ,,*(*grant ,,τ*(*, b*' = 0*,*v*,*

*J grant*



*s*1

*H*

*grJant*

*g*6

*g*5

*g*7

*J J*

*L*

’ *L*

`

*,,*

*stop*

*b* =0 *s*2

ˆ *H*

*sto,p*

{*ackL,* p(l*,* x)*,*

*,,*r *,time*1 = *l*}*l*∈[0*,Tm,ax*] *,*

*sendH , Jack ,*

*g*11 ( *g*8

( *g*10

*b >* 0

*time*2 ∈ [1*,* 2]

*time*2 ≤ *Tmax*

*ls*' = (*ls* + 1)*mod n*

*Jclose* *J*

/’ *J*

*time*' =0

v *ls*

*x*

' = *time*2

/ *,,b*' = *b* − 1*, time*' =0

2

2

*sendL , ,,τ, time*1 ∈ [1*,* 2] *s*3

*time*' = 0 / *b*' = *b* +1

1

*g*9

{*exit, time*2 *> Tmax* ; *J*

'

*Jtime*1 =0

*stop, time*2 *> Tmax* }

Fig. 4. *P ump*1 and *P ump*2

The variable *b* represents the number of messages in the buffer for the high system. In state *g*5, before giving grants, the *Pump*1 empties the buffer.

In state *g*8 *Pump*1 waits for a message from the low system. When the message is received, the automaton *Pump*1 employs a time enclosed in [1*,* 2] to store the message (transition for state *g*9 to state *g*10 ).

Now, *Pump*1 must send an acknowledgment to the low system. The time in which the acknowledgment is sent is probabilistically computed. With *x* we

P

*n*

denote the average time

*i*=1 *xi n*

of the last *n* acknowledgments sent by the high

system, where the management of variables {*x*1*,... , xn*} is owned by *Pump*2.

We denote with *p*(*l, x*) the probability that the NRL Pump sends an acknowl- edgment to the low system at the time *l* when the average acknowledgment delay time is *x*. Of course *p*(*l, x*) can be defined in many ways each of which may yield different performances as of security. We shall examine possible choices for *p*(*l, x*) in Section [5](#_bookmark12). Moreover with the time *Tmax* we denote the NRL Pump timeout.

When the time is expired, *Pump*1 enters state *g*1 and waits for a new connection. If the connection is closed by the low system, then *Pump*1 waits that *Pump*2 has finished its work.

Automaton *Pump*2 starts its execution when the initialization is finished (symbol *grantL*). *Pump*2 can forward a message to the high system when at least one message is in the buffer (condition *b >* 0). After the message has been forwarded (symbol *sendH*), *Pump*2 receives an acknowledgment from the high system (symbol *ackH*). The time of this acknowledgment is stored in the variables *x*1 *... , xn* which are used as an array. The index *ls* represents the position in which the new time must be stored. If the time is over *Tmax*, then the connection is expired. If the connection is closed by the low system and either the buffer is emptied (transition from *s*2 to *s*1) or the time is expired (transition from *s*3 to *s*1), then *Pump*2 enters state *s*1 and waits for a new

connection.

All components can perform actions with the same label. Hence, the whole system *PUMP* is the system

0*.*5

*PUMP* = (*Pump*1||

{*grantL,exit,stop*}

*Pump*2)*.*

The system including also the low system is

1

*A*

*LS* plus *PUMP* = *LS*|| 3 *PUMP*

where *A* = {*exit, close, reqL, validL, grantL*}.

Finally, the system including also the High system is

3

*B*

*LS* plus *PUMP* ||4 *HS*

where *B* = {*exit, close, reqH , validH, grantH*}.

# NRL Pump probabilistic ack delay modeling

The probability *p*(*l, x*), as well as the range of *l*, may be defined in many ways, each of which yields a *probabilistic ack schema*. In the following we consider a few possible scenarios.

* 1. *Uniform distribution*

We may think of computing the NRL Pump ack delay l as follows: l = *x* +

d, where d is a uniformly distributed random variable with range [−Λ*,* +Λ]. Since the expected value of d (notation *E*(d)) is 0, we have *E*(l) = *x* +

*E*(d) = *x*, as required by the NRL Pump specification. As regards *p*(*l, x*) we have: *p*(*l, x*) = if (*l* ∈ [*x* − Λ*, x* + Λ]) then 1*/*(2Λ + 1) else 0. Of course Λ must be chosen small enough so that *x* − Λ ≥ 1.

The drawback of the above approach is that, if the schema is known to the low and high systems, than the following *deterministic* covert channel can be established. To transmit bit *b* (*b* = 0*,* 1) to the low system, the high system sends *h* consecutive ack’s to the pump with delay *Hb*. If *h* is large enough, from the law of large numbers we know that we will have *x* ∼ *Hb*, and *l* ∈ [*Hb* − Λ*, Hb* + Λ]. Without loss of generality, let us assume *H*0 *< H*1. Then, whenever the low system *sees* an ack time *l* ∈ [*H*0 − Λ*, H*1 − Λ) (resp. *l* ∈ (*H*0 + Λ*, H*1 + Λ]), the low system knows (with certainty) that a bit 0 (bit 1) has been transmitted from the high system. Of course, if the ack time is in the interval [*H*1 − Λ*, H*0 + Λ] the low system does not know which bit is being

transmitted from the high system. However, if the high system is sending ack’s with delay *H*0 (*H*1) and *h* is large enough then we know that (with high probability) the low system will observe an ack delay in [*H*0 −Λ*, H*1 −Λ) (resp. (*H*0 + Λ*, H*1 + Λ]). Note that once the low system receives an ack with delay [*H*0 −Λ*, H*1 −Λ) (resp. (*H*0 +Λ*, H*1 +Λ]) then it knows *with certainty* that the high system has sent a bit 0 (bit 1). On the other hand the fact that when the high system is sending ack’s with delay *H*0 (*H*1) the low system will receive an ack with delay [*H*0 − Λ*, H*1 − Λ) (resp. (*H*0 + Λ*, H*1 + Λ]) is highly likely, but not certain.

* 1. *Binomial distribution*

The deterministic covert channel described in the previous section arises since the range of *l* depends on *x*. We overcome this problem by using a *binomial distribution*.

Let *p* ∈ [0*,* 1] and *T* be an integer such that *T* ≥ *Tmax*. Let d be a random variable taking integer values in the range [0*,T* ] with probabilities: *P* (d = *k*)

 

*T*





= *pk*(1 −*p*)*T* −*k*. We note that the range of d does not depend on *p*. Let

*k*

*p* be (*x* − 1)*/T* . Since d has a binomial distribution we have *E*(d) = *T* · *p* =

*T* · *x*−1 = (*x* − 1) and *V ar*(d)= *T* · *p*(1 − *p*)= (*x* − 1)(1 − *x*−1). We can define

*T T*

the NRL Pump ack delay l as follows: l = d + 1, so that l does not depend

on *x*. Then we have *E*(l)= *x*, as required from the NRL Pump specification, and *V ar*(l) = *V ar*(d) = (*x* − 1)(1 − *x*−1).

*T*

Since the range of l does not depend on *x*, the covert channel that we had with the schema used in Section [5.1](#_bookmark13) does not exist. However the high system can send information to the low system as follows. To transmit bit *b* (*b* = 0*,* 1) to the low system, the high system sends *h* consecutive ack’s to the pump with delay *Hb*. If *h* is large enough, from the law of large numbers we know that we will have *x* ∼ *Hb*. The low system can compute a moving average *y* of the last *m* ack delays from the NRL Pump. If *m* is large enough we have *x* ∼ *y*. Then, by comparing *y* with *H*0 and *H*1, the low system can estimate (with arbitrarily low error probability) the bit value sent by the high system. Note that in this case, unlike that in Section [5.1](#_bookmark13), the low system knows the bit sent from the high system only in a probabilistic sense. The error probability depends on many parameters. Here are some of them: *T* (range of l), *h* (high system *sustain time*), *n* (number of samples in NRL Pump moving average), *m* (number of samples in low system moving average), *B* (size NRL

Pump buffer).

To study how the error probability depends on the above parameters we

const -- constant declarations (i.e. model parameters) BUFFER\_SIZE : 5;

NRL\_WINDOW\_SIZE: 5;

LS\_WINDOW\_SIZE: INIT\_NRL\_AVG : INIT\_LS\_AVG : DELTA : HS\_DELAY : DECISION\_THR :

5;

-- size of pump buffer

-- size of nrl pump sliding window

-- size of low system sliding window

4.0; -- init. value of high-syst-to-pump mobile average 0.0; -- init. value of pump-to-low-syst mobile average 10; -- maximal pump-to-low-system ack delay

4.0; -- constant added to high-system-to-pump ack delay

1.0; -- decision threshold

Fig. 5. Constants (model parameters)

will model the NRL Pump using the probabilistic model checker FHP-Mur*ϕ* [[5](#_bookmark37)].

# FHP-Mur*ϕ* Model of the NRL Pump

FHP-Mur*ϕ* (*Finite Horizon Probabilistic Murϕ*) [[5](#_bookmark37),[6](#_bookmark38),[17](#_bookmark48)] is a modified version of the Mur*ϕ* verifier [[7](#_bookmark39),[16](#_bookmark47)]. FHP-Mur*ϕ* allows us to define *Finite State/Discrete Time Markov Chains* and to automatically verify that the probability of reach- ing in *at most k* steps a given error state is below a given threshold.

In this Section we describe our FHP-Mur*ϕ* model of the NRL Pump. We restrict our attention on the NRL Pump features involved in the probabilistic covert channel described in Section [5.2](#_bookmark14). Our goal here is to compute the error probability of the low system when it tries to estimate the bit value sent from the high system. This error probability is a function of *h* (i.e. the number of consecutive high system ack’s to the pump with delay *Hb*, Section [5.2](#_bookmark14)) and some system parameters such as number of samples in high-system-to- pump moving average (registered in a NRL Pump sliding window), number of samples in the pump-to-low-system moving average (registered in a low system sliding window), size of NRL Pump buffer.

FHP-Mur*ϕ* syntax is the same as that of Mur*ϕ*. FHP-Mur*ϕ* comment lines start with --. FHP-Mur*ϕ* can also use C-like comments.

Figure [5](#_bookmark15) shows constants (model parameters) used in our model. FHP- Mur*ϕ* can use (finite precision) real numbers and uses a C-like Csyntax for them.

Figure [6](#_bookmark17) shows types (data structures) used in our model. As to be ex- pected type definition A : B defines A to be an alias for B. The type real(6, 99) denotes finite precision real numbers with 6 decimal digits and a mantissa with absolute value not greater than 99.

Figure [7](#_bookmark18) shows declarations for the global variables of our model. These variables define the state of our model and, therefore, define also the number

type -- type definitions (i.e. data structures)

real\_type : real(6,99); -- 6 decimal digits, |mantissa| <= 99 BufSizeType : 0 .. BUFFER\_SIZE; -- interval [0, BUFFER\_SIZE] NrlWindow : 0 .. (NRL\_WINDOW\_SIZE - 1); -- interval

LSWindow : 0 .. (LS\_WINDOW\_SIZE - 1); -- interval

AckDelay : 0 .. DELTA; -- range of pump-to-low-system ack delay

Fig. 6. Types (data structures)

var -- declarations of global variables (i.e. model state variables) b : BufSizeType; -- number of msgs in pump buffer

nrl\_avg : real\_type; -- high-system-to-pump moving average ls\_avg : real\_type; -- pump-to-low-system moving average nrl\_delays : array[NrlWindow] of real\_type;

-- pump sliding window: last high-system-to-pump ack times ls\_delays : array[LSWindow] of real\_type;

-- low syst sliding window: last pump-to-low-system ack times nrl\_first\_index : NrlWindow; -- cursor nrl sliding window ls\_first\_index : LSWindow; -- cursor low system sliding window nrl\_ack : real\_type; -- pump-to-low-system ack timer

hs\_wait : real\_type; -- high-system-to-pump ack timer ls\_decision: 0 .. 1; -- 0: hs sent 0, 1: hs sent 1

ls\_decision\_state: 0 .. 2; -- 0: to be decided;

-- 1: decision taken; 2: decision taken and state reset done

Fig. 7. Global variables (state variables)

of bytes needed to represent each state (76 bytes in our case). Variable b repre- sents the number of messages in the buffer. Variable nrl avg represents the av- erage of the last delays for the acks received by the pump from the high system. These delays are saved in array nrl delays, where index nrl first index points to the eldest one. Variable ls avg represents the average of the last delays of the acks sent by the pump and received by the low system. These delays are saved in array ls delays, where index ls first index points to the eldest one. Variable nrl ack represents the timer used by the pump for sending the next ack to the low system. Once decided the value of the timer, at each step the nrl ack variable is decreased by 1. The pump sends the ack to the low system when nrl ack ≤ 0. Similarly, variable hs wait is the timer used by the high system for sending acks to the pump. Variables ls decision and ls decision state are used for modeling the transmission of a bit on the covert channel between the high system and the low system.

Mur*ϕ* (and thus FHP-Mur*ϕ*) programming language allows definition of functions and procedures using a Pascal-like syntax. Formal parameters de- clared ”var” are passed by reference. Formals that are not declared ”var” are passed by reference, but the function or procedure is not allowed to modify them.

procedure init(); begin

b := 0;

nrl\_ack := 0;

hs\_wait := 0;

nrl\_avg := INIT\_NRL\_AVG; ls\_avg := INIT\_LS\_AVG; nrl\_first\_index := 0;

ls\_first\_index := 0;

for i : NrlWindow do nrl\_delays[i] := INIT\_NRL\_AVG; end; for i : LSWindow do ls\_delays[i] := INIT\_LS\_AVG; end; ls\_decision := 0;

ls\_decision\_state := 0; end;

Fig. 8. Function init() defines the initial state

Function init() (Figure [8](#_bookmark19)) defines the initial state for our model (i.e. it defines the *initial distribution* for the Markov Chain defined by FHP-Mur*ϕ*). Each variable declared in Figure [7](#_bookmark18) holds the state of a finite state automa- ton. In principle the dynamics of such automata could be described using pictures. However this turns out to be quite complicated. For this reason we just describe such dynamics by giving the transition relation of the automata. Our model of the NRL Pump is organized as the synchronous parallel com- position of many (i.e. those declared in Figure [7](#_bookmark18)) automata. Each transition relation is defined using a procedure which compute the *next state* in the var

formal parameters.

Function nrlpump ack() (Figure [9](#_bookmark20)) defines the transition relation for the automaton modeling the pump-to-low-system ack timer (nrl ack). When the timer reaches 0, the low system gets an ack from the pump. If there is space in the buffer the low system sends a message, and the pump picks a delay d for sending an ack to such message. Delay d is an input of function nrlpump ack(). The value of d is chosen probabilistically using FHP-Mur*ϕ* rules.

Function hs() (Figure [10](#_bookmark21)) defines the transition relation for the automaton modeling the high-system-to-pump ack timer (hs wait).

Function buffer() (Figure [11](#_bookmark22)) defines the transition function for the au- tomaton modeling the NRL Pump buffer. As we can see in Figure [11](#_bookmark22), we have three cases: *i*) both variables hs wait and nrl ack are less than or equal to 0 (pump received ack from the high system and sent ack to the low system), at the same time the pump can send a message to the high system and receive a message from the low system; *ii*) only hs wait is less than or equal to 0 (pump received ack from the high system), the pump can send a message to the high system; *iii*) only nrl ack is less than or equal to 0 (pump sent an

procedure nrlpump\_ack(Var nrl\_ack\_new : real\_type; d : AckDelay); begin

-- ack timer expired, low system sends new msg, timer takes d if ((nrl\_ack <= 0.0) & (b < BUFFER\_SIZE))

then nrl\_ack\_new := d; return; endif;

-- ack timer not expired, waiting ack, timer decremented

if (nrl\_ack > 0.0) then nrl\_ack\_new := nrl\_ack - 1; return; endif;

end;

Fig. 9. Function nrlpump ack defines pump-to-low-system ack times

procedure hs(Var hs\_wait\_new : real\_type); var last\_index : NrlWindow;

begin

-- ack timer not expired, waiting ack, timer decremented if (hs\_wait > 0) then

hs\_wait\_new := hs\_wait - 1; return; endif; -- hs processing msg

-- ack timer expired, high syst receives new msg, timer takes HS\_DELAY if ((hs\_wait <= 0.0) & (b > 0))

then hs\_wait\_new := HS\_DELAY; return; endif;

-- ack timer expired, high system waiting for new msg if ((hs\_wait <= 0.0) & (b <= 0))

then hs\_wait\_new := -2.0; return; endif;

end; -- hs()

Fig. 10. Function hs defines high-system-to-pump ack times

ack to the low system low system), the low system can send a message to the pump.

Function nrlpump() (Figure [12](#_bookmark23)) updates the value of the moving average of the high system ack times. When the pump waits for the ack from the high system (hs wait *>* 0) it updates the value of the last ack delay. When the hs wait timer expires (−1 ≤ hs wait ≤ 0), the pump updates the new average for the acks delays and its auxiliary variables.

Function obs() (Figure [13](#_bookmark24)) defines the transition relation for the automa- ton modeling the low system computation to estimate the high system ack delay. Initially, the low system updates its information regarding the last re- ceived ack delay as done by the pump. When the high system wants to send a bit 0 (1) to the low system, it will use HS DELAY (HS DELAY + 2.0) as ack time. The low system tries to estimate the high system ack time by computing a moving average of the NRL Pump to low system ack times. Such computation

procedure buffer(Var b\_new : BufSizeType); begin

if (((hs\_wait <= 0.0) & (b > 0)) & ((nrl\_ack <= 0.0) & (b < BUFFER\_SIZE)))

then return; endif; -- send and get at the same time, b does not change

if ((hs\_wait <= 0.0) & (b > 0))

then b\_new := b - 1; return; endif; -- high syst gets msg from buffer

if ((nrl\_ack <= 0.0) & (b < BUFFER\_SIZE))

then b\_new := b + 1; return; endif; -- low syst sends msg to buffer

end;

Fig. 11. Function buffer models the pump buffer

procedure nrlpump(Var avg\_new : real\_type); var last\_index : NrlWindow;

begin

last\_index := (nrl\_first\_index + NRL\_WINDOW\_SIZE - 1)%NRL\_WINDOW\_SIZE;

-- high system processing message if (hs\_wait > 0)

then nrl\_delays[last\_index] := nrl\_delays[last\_index] + 1.0; endif;

-- high system sends ack to the pump

if ( (hs\_wait >= -1) & (hs\_wait <= 0.0) ) then avg\_new := nrl\_avg +

((nrl\_delays[last\_index] - nrl\_delays[nrl\_first\_index])/NRL\_WINDOW\_SIZE); nrl\_first\_index := (nrl\_first\_index + 1)%NRL\_WINDOW\_SIZE; nrl\_delays[(nrl\_first\_index + NRL\_WINDOW\_SIZE - 1)%NRL\_WINDOW\_SIZE] := 0; endif

end;

Fig. 12. Function nrlpump updates value of moving average of high syst ack times

is carried out by function obs() which stores its estimation in the global state variable ls avg. The low system goal is to decide if the high system sent a 0 or a 1 bit. This is done using the value held in obs avg. More specifically, if HS DELAY - 1.0 *<* ls avg *<* HS DELAY + 1.0 then obs() decides that the high system sent a bit 0; if HS DELAY + 1.0 *<* ls avg *<* HS DELAY 3.0 then obs() decides that the high system sent a bit 1. To avoid being fooled by noise, obs() takes a decision only when ls avg is stable enough, i.e. when the ab- solute value of the difference (stored in lsdiff) between the present value of ls avg and the previous one is below a given threshold (DECISION THR). As a result the observer may or may not take a decision which in turn may or may not be correct. By the law of large numbers, it is quite clear that waiting

procedure obs(d : AckDelay); var ls\_last\_index : LSWindow; var ackval : real\_type;

var ls\_old : real\_type; var lsdiff : real\_type; begin

if ((nrl\_ack <= 0.0) & (b < BUFFER\_SIZE)) then ackval := d;

ls\_last\_index := (ls\_first\_index + LS\_WINDOW\_SIZE - 1)%LS\_WINDOW\_SIZE; ls\_delays[ls\_last\_index] := ackval;

ls\_old := ls\_avg;

ls\_avg := ls\_avg + ((ls\_delays[ls\_last\_index] -

ls\_delays[ls\_first\_index])/LS\_WINDOW\_SIZE); ls\_first\_index := (ls\_first\_index + 1)%LS\_WINDOW\_SIZE; ls\_delays[(ls\_first\_index + LS\_WINDOW\_SIZE - 1)%LS\_WINDOW\_SIZE] := 0;

-- make decision

if (ls\_decision\_state = 0) then -- decision has not been taken yet

-- make decision only when ls\_avg stable (i.e. lsdiff small) lsdiff := fabs(ls\_avg - ls\_old);

if ( (lsdiff < DECISION\_THR) & (HS\_DELAY - 1.0 < ls\_avg) & (ls\_avg < HS\_DELAY + 1.0) )

then -- decision taken

ls\_decision := 0; ls\_decision\_state := 1; return; endif;

if ( (lsdiff < DECISION\_THR) & (HS\_DELAY + 1.0 < ls\_avg) & (ls\_avg < HS\_DELAY + 3.0) )

then -- decision taken

ls\_decision := 1; ls\_decision\_state := 1; return; endif; endif; endif; end;

Fig. 13. Function obs computes the low syst estimate of the high syst ack time

for a very long time before making a decision (e.g. by making DECISION THR very small) the low system can be quite sure of making a correct decision. However, the more the observer waits for making a decision, the smaller the *bit-rate* of this *covert channel*. Our goal is to estimate the probability that the observer takes the correct decision within *h* time units.

Function goto stop state() (Figure [14](#_bookmark25)) resets the NRL Pump state after a decision about the bit sent from the high system has been made by the low system. This function has nothing to do with the NRL Pump working. It is only used to easy our covert channel measures.

Function main() (Figure [15](#_bookmark26)) updates the system state. Essentially main() triggers the computation of the *next state* of all the automata composing our model.

Function prob delay bin(m, d) (Figure [16](#_bookmark27)) returns the probability that

-- reset all, but obs\_decision\_state procedure goto\_stop\_state(); begin init();

ls\_decision\_state := 2; -- decision taken and reset done end;

Fig. 14. Function goto stop state resets system states after a decision on bit sent has been taken

procedure main(d : AckDelay); Var b\_new : BufSizeType;

Var nrl\_ack\_new : real\_type; Var hs\_wait\_new : real\_type; Var avg\_new : real\_type; begin

if (ls\_decision\_state = 2)

then return; endif; -- decision and state reset done

if (ls\_decision\_state = 0) then -- decision not taken yet b\_new := b;

nrl\_ack\_new := nrl\_ack; hs\_wait\_new := hs\_wait; avg\_new := nrl\_avg; buffer(b\_new); nrlpump\_ack(nrl\_ack\_new, d); obs(d);

nrlpump(avg\_new); hs(hs\_wait\_new); b := b\_new;

nrl\_ack := nrl\_ack\_new; hs\_wait := hs\_wait\_new; nrl\_avg := avg\_new;

else -- decision taken, reset state goto\_stop\_state();

endif; end;

Fig. 15. Function main updates system state

the NRL Pump ack time is d when the NRL Pump moving average value is

m. Function prob delay bin(m, d) implements a binomial distribution with average value m on the interval [0, DELTA].

Figure [17](#_bookmark28) shows the definition of the initial state and of the probabilistic transition rules for our model of the NRL Pump.

The invariant shown in Figure [18](#_bookmark29) states that no decision has been taken. This invariant is false on states in which a decision has been taken. Thus the error probability returned by FHP-Mur*ϕ* allows us to compute the probability *P*dec(*h*) of making a decision within *h* time units.

The invariant shown in Figure [19](#_bookmark30) states that no decision or the correct

decision has been taken. This invariant is false on states in which the wrong

function binomial(n : real\_type; k : real\_type) : real\_type; var result : real\_type;

var nn, kk : real\_type; begin

result := 1; nn := n; kk := k; while (kk >= 1) do

result := result\*(nn/kk); nn := nn - 1; kk := kk - 1;

endwhile;

return (result); end;

function prob\_delay\_bin(m : real\_type; d : AckDelay) : real\_type; var p : real\_type;

begin

p := m/DELTA;

return ( binomial(DELTA, d)\*pow(p, d)\*pow(1 - p, DELTA - d) ); end;

Fig. 16. Function prob delay bin() updates high sys ack timer

-- define init state

startstate "initstate" init(); end;

-- define transition rules ruleset d : AckDelay do rule "time step"

prob\_delay\_bin(nrlavg, d) ==> begin main(d) end; end;

Fig. 17. Startstate and transition rules for NRL pump model

invariant "no\_decision\_taken" 0.0

(ls\_decision\_state = 0);

Fig. 18. Invariant stating that no decision has been taken

invariant "no-dec\_or\_right-dec" 0.0

(ls\_decision\_state = 0) | ((ls\_decision\_state>0) & (ls\_decision=0));

Fig. 19. Invariant stating that no decision or the correct decision has been taken

decision has been taken. Thus the error probability returned by FHP-Mur*ϕ* allows us to compute the probability *P*wrong(*h*) of making a wrong decision within *h* time units.

|  |  |  |
| --- | --- | --- |
| Setting : | 1 | 2 |
| BUFFER SIZE | 2 | 5 |
| NRL WINDOW SIZE | 2 | 5 |
| LS WINDOW SIZE | 2 | 5 |
| cpu time (sec) | 40565.44 | 105866.59s |

Table 1

NRL pump parameter settings and experiments CPU time.

# Experimental Results

In this Section we show our experimental results.

We studied two issues: the probability of making a decision within *h* time units (*P*dec(*h*)) and the probability of making the wrong decision within *h* time units (*P*wrong(*h*)). From the above probabilities we can compute the probability of making the right decision within *h* time units (*P*right(*h*)) as follows: *P*right(*h*) = *P*dec(*h*)(1 − *P*wrong(*h*)).

We studied the above probabilities for various settings of our model param-

eters (defined in Figure [5](#_bookmark15)). More precisely we changed the values of the follow- ing constants: BUFFER SIZE, NRL WINDOW SIZE, LS WINDOW SIZE. In Table [1](#_bookmark32)

we show some of the set of values we considered for the above constants. The last row of Table [1](#_bookmark32) gives the CPU time needed by FHP-Mur*ϕ* to carry out the required analysis (on a 2GHz Intel Pentium PC with Linux OS and 500MB of RAM).

In all cases we considered the probability of making a wrong decision turns out to be identically 0. Thus *P*right(*h*) = *P*dec(*h*).

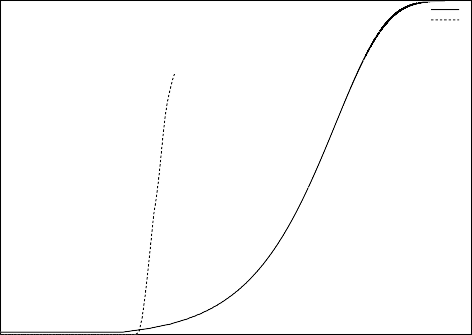
Figure [20](#_bookmark33) shows *P*dec(*h*) (and thus *P*right(*h*)) as a function of *h* for the

parameter settings in Table [1](#_bookmark32).

As can be seen from Figure [20](#_bookmark33), within 10000 time units the low system with probability essentially 1 has decided in the right way the value of the bit sent by the high system.

Our time unit is about the time to transfer messages from/to the pump. Thus we may reasonably assume that a time unit is about 1ms. Then Figure [20](#_bookmark33) tells us that the high system can send bits to the low system at a rate of about 1 bit every 10 seconds, i.e. 0.1 bits/sec. This means that for many applications the NRL Pump can be considered *secure*, i.e. the covert channel has such a low capacity that it would take too long to the high system to send interesting messages to the low system. On the other hand it is not hard to conceive scenarios where also a few bits sent from the high system to the low system can be a security threat.

1



"setting-1.dat" "setting-2.dat"

0.8

0.6

0.4

0.2

0

1 10 100 1000 10000

Fig. 20. Probability of taking a decision within *h* time units.

# Conclusions

We have defined a probabilistic model for the NRL Pump and using FHP- mur*ϕ* we have shown experimentally that there exists a *probabilistic* covert channel whose capacity depends on various NRL Pump parameters (e.g. buffer size, number of samples in the moving average, etc). The experiments show that the probabilistic covert channel exists from the high system to the low system at a rate such that the Pump can be considered secure for many applications.

# References

1. R. Alur, C. Courcoubetis, and D.L. Dill: Verifying Automata Specifications of Probabilistic Real-Time Systems. Proc. REX Workshop, Real-Time: Theory in Practice, Lecture Notes in Computer Science 600, Springer, Berlin, 1992, 28–44.
2. D. Beauquier: On Probabilistic Timed Automata. Theoretical Computer Science 292, 2003, 65–84.
3. D. Bell and L.J. La Padula: Secure Computer Systems: Unified Exposition and Multics Interpretation. Technical report ESD-TR-75-301, MITRE MTR-2997, 1976.
4. A. Bianco and L. de Alfaro: Model Checking of Probabilistic and Nondeterministic Systems. Proc. Int. Conf. on Foundation of Software Technologies and Theoretical Computer Science, Lecture Notes in Computer Science 1026, Springer, Berlin, 1995, 499–513.
5. G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M.V. Zilli: Finite Horizon Analysis of Markov Chains with the Murphi Verifier. Proc. IFIP WG 10.5 Advanced Research Working Conf. on Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science 2841, Springer, Berlin, 2003, 58–71.
6. G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M.V. Zilli: Finite Horizon Analysis of Stochastic Systems with the Murphi Verifier. Proc. Italian Conf. on Theoretical Computer Science, Lecture Notes in Computer Science, Springer, Berlin, 2003.
7. D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang: Protocol Verification as a Hardware Design Aid. Proc. IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors, IEEE Computer Society Press, Los Alamitos, California, 1992, 522–525.
8. P.R. Halmos: Measure Theory. Springer, Berlin, 1950.
9. H. Hansson and B. Jonsson: A Logic for Reasoning About Time and Probability. Formal Aspects of Computing 6, 1994, 512–535.
10. M.H. Kang, J. Froscher, and I.S. Moskowitz: A Framework for MLS Interoperability. Proc. IEEE High Assurance Systems Engineering Workshop, IEEE Computer Society Press, Los Alamitos, California, 1996, 198–205.
11. M.H. Kang, J. Froscher, and I.S. Moskowitz: An Architecture for Multilevel Security Interoperability. Proc. IEEE Computer Security Application Conf., IEEE Computer Society Press, Los Alamitos, California, 1997, 194–204.
12. M.H. Kang, A.P. Moore, and I.S. Moskowitz: Design and Assurance Strategy for the NRL Pump. IEEE Computer 31, 1998, 56–64.
13. M.H. Kang and I.S. Moskowitz: A Pump for Rapid, Reliable, Secure Communication. Proc. ACM Conf. on Computer and Communication Security, ACM Press, New York, 1993, 119–129.
14. M.H. Kang, I.S. Moskowitz, and D. Lee: A Network Pump. IEEE Transactions on Software Engineering 22, 1996, 329–338.
15. M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston: Automatic Verification of Real-Time Systems with Discrete Probability Distribuitions. Proc. AMAST Workshop on Real-Time and Probabilistic Systems, Lecture Notes in Computer Science 1601, Springer, Berlin, 1999, 79-95.
16. Murphi web page: <http://sprout.stanford.edu/dill/murphi.html>
17. Cached Murphi web page: <http://www.dsi.uniroma1.it/>∼tronci/cached.murphi.html