Electronic Notes in Theoretical Computer Science 130 (2005) 101–128 

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

A New Method for Transforming Timed Automata

# Ahmed Khoumsi[1](#_bookmark0) Lucien Ouedraogo[2](#_bookmark0)

*Dept. of Elect. and Comp. Eng.*

*University of Sherbrooke Sherbrooke J1K2R1, CANADA*

Abstract

Discrete events systems (DES) are defined by the sequences of events they can execute. For exam- ple, communication protocols and computer networks can be seen as DES. Finite state automata (FSA) are convenient for *studying* (i.e., analyzing, designing) DES, and timed automata (TA) are convenient for *describing* real-time DES. An approach for studying a real-time DES, has been to transform a TA describing the real-time DES into an equivalent FSA, and then to study the latter. We propose here a new transformation method of TA into FSA. The method is well suited for conformance testing and supervisory control of real-time DES.

*Keywords:* Transformation, Timed Automata, Set-Exp-Automata.

# Introduction

Discrete events systems (DES) interact with their environment by executing events. Here are some examples of DES: communication protocols (events: send message, receive message, ... ), telephone systems (events: hang on, hang

up, ... ), mobile robot (events: start moving, stop, ... ). Correctness of a Real- time DES (RTDES) depends not only on *how* but also on *when* the RTDES interacts with its environment. Timed Automata (TA) [[4](#_bookmark45)] are convenient to describe RTDES, but a TA has *inﬁnitely* many states. An approach to give a finite representation of the state space of a TA, is to generate a region

1 Email: [Ahmed.Khoumsi@USherbrooke.ca](mailto:Ahmed.Khoumsi@USherbrooke.ca)

2 Email: [Lucien.Ouedraogo@USherbrooke.ca](mailto:Lucien.Ouedraogo@USherbrooke.ca)

1571-0661 © 2005 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.2005.03.007

automaton (RA) [[2](#_bookmark42)]. The state space of a RA is finite but can suffer from state explosion [[2](#_bookmark42)]. To reduce such a state explosion, several *minimization* methods have been proposed to transform a TA into an automaton with much less states than the corresponding RA [[3](#_bookmark43),[15](#_bookmark55),[5](#_bookmark46),[6](#_bookmark47),[14](#_bookmark56),[1](#_bookmark41)].

We propose a new method called *SetExp* for transforming a TA into a finite state automaton (FSA), called Set-Exp-Automaton (SEA), which uses two additional types of events: *Set* and *Exp* that model the setting and expiring of clocks, respectively. A TA and the corresponding SEA represent two different ways to specify the same order and timing constraints of events.

In comparison to other minimization methods, *SetExp* is well suited for *conformance testing* and *supervisory control* of RTDES [3](#_bookmark2) . Conformance test- ing aims at *checking* whether an implementation conforms to a specifica- tion [[13](#_bookmark54)], and supervisory control aims at *forcing* an implementation to con-

form to a specification [[12](#_bookmark53)]. The application of *SetExp* to conformance testing (resp., supervisory control) has been demonstrated in [[7](#_bookmark48),[10](#_bookmark51)] (resp., in [[11](#_bookmark52),[8](#_bookmark49),[9](#_bookmark50)]). In [[7](#_bookmark48),[10](#_bookmark51),[11](#_bookmark52),[8](#_bookmark49),[9](#_bookmark50)], *SetExp* is used as a black-box and the focus is on its applica- tion. In the present study, our aim is to “see inside *SetExp*”; more precisely, we explain formally how *SetExp* can be implemented. Therefore, this paper

is complementary to [[7](#_bookmark48),[10](#_bookmark51),[11](#_bookmark52),[8](#_bookmark49),[9](#_bookmark50)].

The remainder of the paper is structured as follows. Sect. [2](#_bookmark1) introduces the TA model. In Sect. [3](#_bookmark5), we illustrate *SetExp* by simple examples, and we present the SEA model. Sects. [4](#_bookmark8) and [5](#_bookmark14) explain in a formal way how *SetExp* is realized. In Sect. [6](#_bookmark31), we present a theorem that states correctness and some properties of *SetExp*. And Sect. [7](#_bookmark40) concludes the paper.

# Timed Automata (TA)

A clock is a real-valued variable that can be reset (to 0) at the occurrence of an event and such that, between two resets, its derivative (w.r.t. time) is equal to 1. Let C = {*c*1*,* ··· *, cNc* } be a set of clocks.

A clock constraint is a formula in the form “*ci* ∼ *k*”, where *ci* is a clock,

∼∈ {*<, >,* ≤*,* ≥*,* =} and *k* is a nonnegative integer. Let ΦC denote the (infinite) set of clock constraints depending of clocks of C.

And let 2*X* denote the set of subsets of a set *X*.

3 *SetExp* does not aim at competing with other minimization methods in *Veriﬁcation*.

* 1. *Syntax of TA*

A TA is defined by (L*,* A*,* C*,* T *, l*0), where: L is a finite set of locations, *l*0 is the initial location, A is a finite set of events (*alphabet*), C is a finite set of clocks, and T ⊆ L×A× L× 2ΦC × 2C is a transition relation. A transition of TA is thus defined by *T* = ⟨*q*; *σ*; *r*; *G*; *Z*⟩, where: *q* and *r* are origin and destination locations, *σ* is an event, *G* is a finite subset of ΦC and is called *guard* of *T* , and *Z* is a subset of C and is called *reset* of *T* .

The example of Fig. [1](#_bookmark3) illustrates this definition. Locations are represented by nodes, and a transition Tr = ⟨*q*; *σ*; *r*; *G*; *Z*⟩ is represented by an arrow linking *q* to *r* and labeled (*σ*; *G*; *Z*). An empty *G* or *Z* is represented by “-”.



l 0

Tr1

Tr3

( ; − ;{ c1 , c2} )

Tr4

( ; { c1> 1, c2 < 2} ; − )

l1

Tr2

( ;{c1 < 3}; { c1})

l 2

(  ; {c1> 1,c2> 2} ; − )

Fig. 1. Example of TA

* 1. *Semantics of TA*

At time *τ*0 = 0, the TA *A* = (L*,* A*,* C*,* T *, l*0) is at location *l*0 with all clocks of C equal to 0. A transition Tr=⟨*q*; *σ*; *r*; *G*; *Z*⟩ is *enabled* when *q* is the current location and *all* the clock constraints of the guard *G* (if any) evaluate to *true*; otherwise, Tr is *disabled* . From this location *q*, the event *σ* is executed only when Tr is enabled; and after the execution of *σ*, location *r* is reached and the clocks in *Z* are reset.

For example, the TA of Fig. [1](#_bookmark3) is initially in location *l*0; it reaches *l*1 at the occurrence of *σ*. From location *l*1, the TA reaches *l*2 at the occurrence of *µ*. From location *l*2, the TA reaches *l*0 at the occurrence of *φ* or *ρ*. Let *δu,v* denote the delay between events *u* and *v*. We have: *δσ,µ* ≤ 3, *δσ,φ <* 2, *δσ,ρ* ≥ 2, *δµ,φ* ≥ 1, and *δµ,ρ* ≥ 1.

The semantics of a TA *A* can also be defined by the set of timed sequences accepted by *A*. a A *timed sequence* of a TA is a sequence “(*e*1*, τ*1) ··· (*ei, τi*) ·· ·”, where *e*1*,* ··· *, ei* ∈ A, each *τi* is the time of occurrence of *ei*, and 0 *< τ*1 *<*

··· *< τi <* ·· ·. Now given a TA *A*, let us define the acceptance by *A* of a timed sequence *λ* = (*e*1*, τ*1)(*e*2*, τ*2) ·· ·. Let *n* be the length of *λ* (*n* can be infinite), and *λi* = (*e*1*, τ*1) ··· (*ei, τi*) be the prefix of *λ* of length *i*, for 0 ≤ *i* ≤ *n* (*i* is finite). *λ is accepted by A iff* :

* Either *λ* is the empty sequence *λ*0;
* Or *A* has a sequence of length *n* of consecutive transitions *Tr1 Tr2* ··· that starts at *l*0 and s.t. ∀*i* = 1*,* 2*,* ··· *, n*: the event of *Tri* is *ei* and, after the execution of *λi*−1, *Tri* is enabled at time *τi*.

Definition 2.1 *The* Timed language of a TA *A (TLTA) is the set of timed sequences accepted by A. That is, TLTA models the behavior of A.*

*A*

*A*

A TA allows to express constraints on delays between events. For example, to specify that the delay *d* between two transitions *Tr1* and *Tr2* is such that *d* ∈ [1*,* 3], we use a clock *c*1 as follows: the reset of *Tr1* is {*c*1}, and the guard of *Tr2* is {(*c*1 ≥ 1)*,* (*c*1 ≤ 3)}.

The class of TA that we will consider respects the following hypothesis:

Hypothesis 2.1 *Each TA is assumed deterministic, i.e., if Tr1 and Tr2 are two transitions executing the same event from the same location, and are en- abled at the same time, then they lead to the same location and reset the same clock(s).*

# Illustration of *SetExp*, Set-Exp-Automata (SEA)

The aim of this paper is to present a method *SetExp* for transforming a TA into a finite state automaton, called Set-Exp-Automaton (SEA), which uses two additional types of events: *Set* and *Exp* that model the setting and expiring of clocks, respectively. A TA and the corresponding SEA represent two different ways to specify the same order and timing constraints of events. In this section,

we first introduce the events *Set* and *Exp*, followed by a trivial example that illustrates *SetExp*. Then, we present the SEA model, followed by a running example of *SetExp* corresponding to the TA of Fig. [1](#_bookmark3).

* 1. *Events Set and Exp*

An event *Set*(*ci* ; *k* ) means: clock *ci* is reset (to 0) and will expire when its value is equal to *k*. And *Set*(*ci* ; *k1 , k2 ,* ··· *, kp*) means that *ci* is reset and will expire several times, when its value is equal to *k*1*, k*2*,* ··· *, kp*, respectively. We assume without loss of generality that *k*1 *< k*2 *<* ··· *< kp*.

An event *Exp*(*ci* ; *k* ) means: clock *ci* expires and its current value is *k*. Therefore, *Set*(*ci* ; *k* ) is followed (after a delay *k*) by *Exp*(*ci* ; *k* ), and

*Set*(*ci* ; *k1 , k2 ,* ··· *, kp*) is followed (after delays *k*1*,* ··· *, kp*) by *Exp*(*ci* ; *k1* )*,*

*Exp*(*ci* ; *k2* )*,* ··· *, Exp*(*ci* ; *kp*). When a *Set*(*ci* ; ∗) occurs, then all expirations of *ci* which were foreseen before this *Set*(*ci* ; ∗) are canceled.

* 1. *Trivial example of transformation*

To explain the intuition of *SetExp*, let us consider the following two speci- fications. Specification 1: a task must be realized in less than two units of time. Specification 2: at the beginning of the task an alarm is programmed so that it occurs after two time units, and the task must be terminated before the

alarm. It is clear that these two expressions define the same timing constraint. In this example, *SetExp* can be used to obtain the second specification from the first one. The programming of the alarm corresponds to a *Set* event, and the occurrence of the alarm corresponds to an *Exp* event.

* 1. *Transitions of SEA*

Let *A* be a TA and *SetExp*(*A*) be the corresponding SEA. In the following: *σ* denotes an event of the alphabet of the TA *A*, S (resp. E) denotes a set of *Set* (resp. *Exp*) events, and occurrence of S (resp. E) means the simultaneous occurrences of all the events in S (resp. E). Let us categorize the transitions of the SEA *SetExp*(*A*) into three types:

Type 1 : a transition labeled (E) represents the occurrence of E.

Type 2 : a transition labeled (*σ*) or (*σ,* S): (*σ*) represents the occurrence of *σ*, and (*σ,* S) represents the simultaneous occurrences of *σ* and S. A transition TR of Type 2 in the SEA *SetExp*(*A*) corresponds to a transition of *A*.

Type 3 : a transition labeled (E *, σ*) or (E *, σ,* S): (E *, σ*) represents the simul- taneous occurrences of E and *σ*, (E *, σ,* S) represents the simultaneous oc- currences of E, *σ* and S. A transition TR of Type 3 in the SEA *SetExp*(*A*) corresponds to the simultaneous executions of E and a transition Tr of *A*.

* 1. *Syntax and semantics of SEA*

A SEA *B* is a particular FSA and its syntax can be simply defined by (*Q,* Γ*, δ, q*0), where *Q* is a set of states, *q*0 is the initial state, and Γ is an alphabet consisting of labels of transitions of types 1, 2 or 3. *δ* ⊆ *Q* × Γ × *Q* is a set of transitions.

Let a *sequence* denote a (finite or infinite) sequence “*E*1 ··· *Ei* ·· ·”, where *Ei* ∈ Γ. From a behavioral point of view, let us define the semantics of a SEA *B* = (*Q,* Γ*, δ, q*0) by the set of sequences accepted by *B*:

* A finite sequence *µn* = *E*1 ··· *En* is accepted by *B iff* it labels a sequence of *n* consecutive transitions of *B* that starts in *q*0 and terminates in a state with no outgoing transition of Type 1 or 3. Intuitively, *B* can execute *µn*

and then stops only if there is no expiration after *µn*. This is so, because expirations cannot be delayed.

* An infinite sequence *µ*∞ is accepted by *B iff* it labels a sequence of consec- utive transitions of *B* that starts in *q*0.

We can now introduce the notion of Language of a SEA:

Definition 3.1 *The language of a SEA B (LSEA) consists of the (ﬁnite and inﬁnite) sequences accepted by B. That is, LSEA models the behavior of B.*

*B*

*B*

Given a SEA *B*, note that *LSEA* implicitly respects the following condi- tion, called the *Consistency condition*: every *Set*(*c*; *k* ) and its corresponding *Exp*(*c*; *k* ) are effectively separated by time *k*.

*B*

* 1. *Running example*

For the TA *A* of Fig. [1](#_bookmark3), we obtain the SEA *SetExp*(*A*) of Fig. [2](#_bookmark11). We will explain in Sect. [5](#_bookmark14) how this SEA is computed. Transitions of type 1 are those labeled *Exp*(*c1* ; *1* ), *Exp*(*c1* ; *3* ), *Exp*(*c2* ; *2* ) and (*Exp*(*c1* ; *1* )*, Exp*(*c2* ; *2* )). Tran- sitions of Type 2 are those labeled (*σ, Set*(*c1* ; *3* )*, Set*(*c2* ; *2* )), (*µ, Set*(*c1* ; *1* )), *φ* and *ρ*; they correspond to transitions Tr1, Tr2, Tr3, and Tr4, respectively, of the TA *A* of Fig. [1](#_bookmark3). Transitions of Type 3 are those labeled

(*Exp*(*c2* ; *2* )*, σ, Set*(*c1* ; *3* )*, Set*(*c2* ; *2* )), (*Exp*(*c1* ; *1* )*, φ*),

(*Exp*(*c1* ; *1* )*, ρ*), (*Exp*(*c2* ; *2* )*, ρ*), (*Exp*(*c1* ; *1* )*, Exp*(*c2* ; *2* )*, ρ*),

(*Exp*(*c2* ; *2* )*, µ, Set*(*c1* ; *1* )), (*Exp*(*c1* ; *3* )*, µ, Set*(*c1* ; *1* )). Finite sequences ac-

cepted by the SEA *SetExp*(*A*) (i.e., ∈ *LSEA* ) start in *q*0 and terminate in

*SetExp*(*A*)

a shaded state (i.e., state without outgoing *Exp* event).

*SetExp*, that transforms a TA *A* = (L*,* A*,* C*,* T *, l*0) into a SEA *SetExp*(*A*), is realized in two steps. Step 1 does not modify the structure of *A*: it replaces the resets of transitions by *Set* events (substep 1a), and rewrites the guards of transitions in the form of order constraints relatively to *Exp* events (sub- step 1b). Step 2 generates *SetExp*(*A*) that describes explicitly all the possible orders of events, including *Set* and *Exp* events in addition to the events of the alphabet of *A*. Steps 1 and 2 are presented in Sects. [4](#_bookmark8) and [5](#_bookmark14), respec- tively. The results of Steps 1 and 2 will be denoted *A*' = *StepOne*(*A*) and *B* = *SetExp*(*A*) = *StepTwo*(*A*'), respectively.

1. First step of *SetExp*

Definition 4.1 *In a TA A: a* path *is a transition or sequence of consecutive transitions of A.*

Definition 4.2 *In a TA A: a transition T* ' *is said* reachable *from a transition*



 Set(c 1, 3) Set(c 2, 2)

q 0

q 1

Exp(c , 2)

2

q 7

Exp(c , 3)

1

q 8

1

 Set(c 1, 3) Set(c 2, 2)

Exp(c 2, 2)  Set(c 1, 3) Set(c 2, 2)

 Set(c , 3) Set(c 2, 2)



Set(c 1, 1)

Exp(c 2, 2)



Set(c 1, 1)

Exp(c , 3)

q

2



Set(c 1, 1)

 1

Set(c , 1)

1

Exp(c 1, 1)

Exp(c 1, 1)

Exp(c 2, 2)



q1

5

q 2

5

q 3

5

q 4

5

q 3

Exp(c , 1)

1

Exp(c 2, 2)

Exp(c 1, 1) Exp(c 1, 1)

Exp(c 1, 1)



Exp(c , 1)

1

Exp(c 2, 2)

Exp(c , 1)

Exp(c 1, 1)



1 

Exp(c , 1)

q

1 

q 4

6

Exp(c 1, 1) Exp(c 1, 1)



Exp(c 2, 2)



Exp(c , 2)





2 Exp(c , 2)

2

q’0

Fig. 2. Example of SEA

*T* iff *A contains a path T* ··· *T* '*.*

Remark 4.1 *In Def.* [*4.1*](#_bookmark9) *the guards of transitions are not taken into account, and thus, a path can be* unexecutable*. Therefore in Def.* [*4.2*](#_bookmark10)*, “reachability of transition” is actually an abuse of the language. We need this non-standard deﬁnition to deﬁne in a simple way Substep 1a (in Sect.* [*4.1*](#_bookmark13)*).*

Notations 4.1 *In a TA A:*

* ঩*T denotes the set of transitions that are reachable from a transition T .*
* *℘T* →*T* ' *denotes the set of paths between two transitions T and T* '*,* excluding *T and T* '*. That is, ℘T* →*T* ' *contains every path T*1 ··· *Tm such that: the destination location of T is the origin location of T*1*, and the origin location of T* ' *is the destination location of Tm.*
* *ZP is the set of clocks that are reset in a path P.*
* *ZT denotes the set of clocks that are reset in T .*
* *GT denotes the guard of T .*

Since Step 1 replaces the resets of transitions by *Set* events and rewrites the guards of transitions in the form of order constraints relatively to *Exp* events, we will use the following Notations [4.2](#_bookmark12):

Notations 4.2 *For a transition T of a TA A:*

* Z*T is the set of Set events associated to T , and*

Z*T* (*ci* ) *is the part of* Z*T using ci.*

* G*T is the rewritten guard associated to T , and*

G*T* (*ci* ) *is the part of* G*T using ci.*

Notations 4.3 *In a formal expression: Symbol “*:=*” denotes an assignment;* *and Symbol “*:*” have two possible meanings:*

* *In a deﬁnition of a set,* {*A* : *B*} *denotes the set of A such that B is satisﬁed. Example:* {(*x, y*) : *x > y*} *is the set of pairs* (*x, y*) s.t. *x > y.*
* *An expression “*∀*A* : *B” means that B is applied whenever A is satisﬁed. Example:* ∀*x >* 3 : *A*[*x*] := 0 *means that A*[*x*] *is set to zero* ∀*x >* 3*.*
  1. *Substep 1a: replacing clock resets by Set events*

In a TA, a clock *c* is reset with the objective to compare later its value to (at least) one constant, say *k*. The event *Set*(*c*; *k* ) is very convenient for that purpose, because it resets *c* and programs *Exp*(*c*; *k* ) which is a notification when *c* = *k*. The aim of Substep 1a is indeed to replace by *Set*(*ci* ; *k* ), each reset of *ci* if it is compared later to *k* before a subsequent reset. If for the same transition, we obtain several *Set*(*ci* ; *k1* )*, Set*(*ci* ; *k2* )*,* ·· ·, we replace them by a single *Set*(*ci* ; *k1 , k2 ,* ·· ·). Without loss of generality, we consider that *k*1 *< k*2 *<* ·· ·. Here is a formal definition of Substep 1a:

∀*T* ∈ T *,* ∀*ci* ∈ C : Z*T* (*ci* ) := {*Set*(*ci* ; *U* ) :

*ci* ∈ *ZT ,U* ={*k* : ∃*T* ' ∈ ঩*T ,* ∃(*ci* ∼ *k*) ∈ *GT*' *,* ∃*P* ∈ *℘T*→*T*' *, ci* /∈ *ZP* }}

∀*T* ∈ T : Z*T* := ∪*ci*∈C Z*T* (*ci* )

Note that Z*T* (*ci* ) is either *empty* if *ci* /∈ *ZT* , or a *singleton* {*Set*(*ci* ; ∗)} if

*ci* ∈ *ZT* .

* 1. *Substep 1b: rewriting the guards*

Every guard *ci* ∼ *k* is rewritten in the form ∼ *Exp*(*ci* ; *k* ). Formally:

∀*T* ∈ T *,* ∀*ci* ∈ C : G*T* (*ci* ) := {∼ *Exp*(*ci* ; *k* ) : (*ci* ∼ *k*) ∈ *GT* }

∀*T* ∈ T : G*T* := ∪*ci*∈C G*T* (*ci* )

If we apply Step 1 to the TA *A* of Fig. [1](#_bookmark3), we obtain the *StepOne*(*A*) of Fig. [3](#_bookmark15). Labels are represented in one or two lines. The first line contains the event of the transition and *Set* event(s) if any. The second line contains the rewritten guard(s) if any.





> Exp(c 1, 1) <Exp(c 2, 2)

Tr3

l

0

Tr1

 Set(c 1, 3) Set(c 2, 2)



> Exp(c 1, 1) >Exp(c 2, 2)

l1

 Set(c 1, 1)

< Exp(c1 , 3)

Tr2

l 2

Tr4

Fig. 3. Step 1 applied to the TA of Fig. [1](#_bookmark3)

1. Second step of *SetExp*

Step 2 transforms *A*' = *StepOne*(*A*) into a SEA *B* = *SetExp*(*A*) = *StepTwo*(*A*'). *B* describes explicitly all the possible sequences of events, in- cluding *Set* and *Exp* events in addition to the events of the alphabet of *A*.

* 1. *States of SEA*

Each state of a SEA is defined by a location and inequation(s) of clock(s). The latter definition, which we call *state deﬁnition* and present in this subsection, is not necessary for defining the syntax and semantics of a given SEA; but it is useful for algorithmic purpose, during the construction of *SetExp*(*A*) from *StepOne*(*A*). Once *SetExp*(*A*) constructed, this state definition can be ignored. Here are a few necessary definitions.

Definition 5.1 *Clock ci is* active *when at least an expiration of ci is expected. More precisely, while a Set*(*ci* ; *k1 ,* ··· *, kp*) *is the last Set*(*ci* ; ∗) *that has oc- curred: ci is* active iff *Exp*(*ci* ; *kp*) *has not occurred. Otherwise, ci is* inactive*.*

Note that the set of current active clocks depends on the current time and on the history of the automaton up to the current time.

Definition 5.2 *For a Set* (*ci* ; *k1 ,* ··· *, kp*)*, a* Clock-Cond *is an expression in one of the following three forms, for* 1 ≤ *u < p:*

*“*0 *< ci < k*1*”, which holds between Set*(*ci* ; *k1 ,* ··· *, kp*) *and Exp*(*ci* ; *k1* )*; “k**u < ci < ku*+1*”, which holds between Exp*(*ci* ; *ku*) *and Exp*(*ci* ; *ku*+*1* )*; and “kp < ci”, which holds after Exp*(*ci* ; *kp*)*.*

Definition 5.3 *For a Set* (*ci* ; *k1 ,* ··· *, kp*)*, an* ExpSeq *is a sequence kuku*+1 ··· *kp, for* 1 ≤ *u* ≤ *p; it speciﬁes:*

* *if u >* 1*: the remaining values to which ci will expire after Exp*(*ci* ; *ku*−*1* )*;*
* *if u* = 1*: the values to which ci will expire after Set*(*ci* ; *k1 , k2 ,* ··· *, kp*)*.*

Definition 5.4 *A* ∆Clock-Cond *is an expression in one of the following four forms, where m, m*1*, m*2 *are integers (they can be positive, negative, or null):*

*“ci* − *cj < m*2*”, “m*1 *< ci* − *cj”, “m*1 *< ci* − *cj < m*2*”, and “ci* − *cj* = *m”.*

Let *A* be a TA and *B* = *SetExp*(*A*) be the corresponding SEA to be computed. A *state deﬁnition* of a state *q* of *B* consists of three parts:

Part 1 indicates the location of *A* that corresponds to *q*, and is noted *Lq* .

Part 2 consists of a Clock-Cond and an ExpSeq for each clock in C. More precisely, for each *ci*, Part 2 contains (*Cq*(*ci* )*, Kq* (*ci* )) which is in one of the following forms, considering that *Set*(*ci* ; *k1 , k2 ,* ··· *, kp*) is the last *Set* event of *ci* before *q* is reached:

* If *q* is reached before *Exp*(*ci* ; *k1* ):

*Cq* (*ci* ) = (0 *< ci < k*1), *Kq*(*ci* ) = (*k*2*k*3 ··· *kp*).

* If *q* is reached between *Exp*(*ci* ; *ku* ) and *Exp*(*ci* ; *ku*+*1* ), *u* = 1*,* ··· *,p* − 2:

*Cq* (*ci* ) = (*ku < ci < ku*+1), *Kq*(*ci* ) = (*ku*+2 ··· *kp*).

* If *q* is reached between *Exp*(*ci* ; *kp*−*1* ) and *Exp*(*ci* ; *kp*):

*Cq* (*ci* ) = (*kp*−1 *< ci < kp*), *Kq* (*ci* ) = *ϵ* (i.e., it is empty).

* If *q* is reached after *Exp*(*ci* ; *kp*): *Cq* (*ci* ) = (*kp < ci*), *Kq* (*ci* ) = *ϵ*.

Note that *ci* is active in the three first cases, and inactive in the fourth case.

Part 3 consists of zero or one ∆Clock-Cond ∆*Cq* (*ci , cj* ) for each pair of clocks

*ci, cj* ∈ C.

Notations 5.1 *To recapitulate, a state q of B* = *SetExp*(*A*) *is deﬁned by three parts, which we note q* = ⟨*Lq , Cq,* ∆*Cq* ⟩*. Lq is a location; Cq is a Nc- tuple of pairs* (*Cq*(*ci* ); *Kq* (*ci* ))*, for i* = 1*,* ··· *, Nc, where Cq*(*ci* ) *is a Clock-Cond and Kq* (*ci* ) *is an ExpSeq; and* ∆*Cq is a set of* ∆*Clock-Conds.*

The usefulness of the three parts can be explained as follows:

* The location *Lq* is used to determine the transitions of *A* that are candidate in *q*, which are the outgoing transitions of *Lq* in *A*.
* Clock-Conds of *q* are used to determine:

-the transitions of *A* that are enabled in *q* (among the candidate ones), and

-the expected next expiration (if any) of each clock.

* ExpSeqs of *q* are used to determine the Clock-Conds and ExpSeqs of the state reached by each transition of *A* that is enabled in *q*.
* ∆Clock-Conds of *q* are used to determine the expirations (or set of simul- taneous expirations) that can *really* occur at *q*, among the expected ones determined from Clock-Conds of *q*. Therefore when all the clocks are inac- tive, ∆Clock-Conds are irrelevant, and thus, Part 3 is empty.

When an expiration (or set of expirations) can occur in *q*, we obtain an enabled transition of type 1. When an event *σ* labelling a transition of *A* can occur in *q*, we obtain an enabled transition of type 2. And when expiration(s) and a *σ*

can occur simultaneously, we obtain an enabled transition of type 3.

The state definition for the SEA of Fig. [2](#_bookmark11) is represented in Fig. [4](#_bookmark17). ExpSeqs are empty, and thus not represented, because in every *Set*(*ci* ; ∗), ∗ consists of a single value.

Fig. 4. Example of SEA with state definition



q 1

q 7

q 8

l0 0 < c2

0 < c1

 Set(c 1, 3) Set(c 2, 2)

l1 0 < c2 < 2

0 < c1 < 3 Exp(c , 2)

2

0 < c1 < 3 Exp(c , 3)

3 < c 1

q

 Set(c , 3) Set(c 2, 2)

0

−

1

 Set(c 1, 3) Set(c 2, 2)

c1 − c2 = 0

l1 2 < c2

c1 − c2 = 0

1

l1

2 < c 2

−

Exp(c 2, 2)  Set(c 1, 3) Set(c 2, 2)



Set(c 1, 1)

Exp(c 2, 2)



Set(c 1, 1)

Exp(c , 3)

 1

q 2

l 0 < c2 < 2

0 < c1 < 1



Set(c 1, 1)

Set(c , 1)

1

2

Exp(c 1, 1)

0 < c2 − c1 < 2

Exp(c 2, 2)

Exp(c 1, 1)

q



5

0 < c < 1

q1

3

l 2

2 1 c2

5

0 < c < 1

q 2

q 3

q 4

<

l 2

2 <1 c2 c2 − c1 = 2

Exp(c 1, 1)

l

2

2 <1 c2

2 < c2 − c1 < 3

5

0 < c < 1

5

0 < c < 1

l

2

l 2 0 < c2 < 2 0 < c2 − c1 < 1

1 < c1

1 < c2 − c1 < 2

2 <1 c2 c2 − c1 = 3

Exp(c , 1)

1

Exp(c 2, 2)

Exp(c 1, 1)

Exp(c 1, 1)



Exp(c , 1)

Exp(c 1, 1)

1 

q

1

Exp(c 2, 2)

Exp(c , 1)



1 < c1

Exp(c , 1)

4

1 < c

l 2 2 < c2

1 

l0 0 < c2 < 2 0 < c2 − c1 < 1

1

− q 6

Exp(c 1, 1) Exp(c 1, 1)



Legend

Exp(c 2, 2)



Exp(c , 2)



q’0



2 Exp(c , 2)

1 < c 1

2 < c 2

−

Clock−Condition(s)

2

l 0

Clock−Condition(s)

location

* 1. *Principle of construction of SEA*

Let *A*' = *StepOne*(*A*) be the automaton computed in the first step (see Sect. [4](#_bookmark8)). The principle of construction of *B* = *StepTwo*(*A*') from *A*' is as follows:

* First of all, we construct its initial state *q*0 = ⟨*l0 , C0 ,* −⟩ where: *l*0 is the initial location of *A*', *C0* is the *Nc*-tuple of pairs (0 *< ci*; *ϵ*), for *i* = 1*,* ··· *, Nc*. “0 *< ci*” and *ϵ* are a Clock-Cond and an empty ExpSeq, respectively; they mean that *ci* has been reset and no expiration is expected, i.e., *ci* is initially inactive. There is no ∆Clock-Cond because all clocks are inactive.
* Then, we proceed iteratively as follows until no new state or transition is constructed: For every constructed state *q*, we determine every event or set of (simultaneous) events that can label a transition (of type 1, 2 or 3) enabled in *q*. And for every determined (enabled) label *l*, we construct: the state *r* reached by *l* from *q*, and the corresponding transition.

We see that this iterative process requires to determine if a label is enabled in a state, and to compute the state reached by an enabled label and the corresponding transition. Here are some related notations and definitions:

Notations 5.2 *(OUT* (*l* )*, EXP* (*q* )*, ) :*

* 1. *For a location l of A*' *(*= *StepOne*(*A*)*), OUT* (*l* ) *denotes the set of labels of the outgoing transitions of l, including the empty label noted .*
  2. *If in a state q of SEA, cu*1 *, cu*2 *,* ··· *, cup are the active clocks and Exp*(*cui* ; *kui* )

*is the expected next expiration of cui , for i* = 1*,* ··· *, p, then: EXP* (*q* ) = {*Exp*(*cu1* ; *ku1* )*,* ··· *, Exp*(*cup* ; *kup* )}*.*

Let *q* be a state of the SEA *B* = *StepTwo*(*A*') and *l* ∈ (*2 EXP*(*q*) ×

*OUT* (*Lq* )) \ {(∅*,* )} be a label of a transition of type 1, 2 or 3.

Definition 5.5 *l* is said to be enabled in *q, which we note* (*q* →*l* )!*,* iff *it is possible that at a same time in q,* all *the events of l occur simultaneously before any other event.*

Definition 5.6 *Assuming* (*q* →*l* )!*, we can deﬁne the state reached from q after* *the occurrence of l, which we note* (*q* →)*.*

*l*

In Sect. [5.3](#_bookmark19), we explain in a formal way how to determine whether (*q* →*l* )! and how to compute the state (*q* →*l* ). Subsection [5.3.1](#_bookmark21) (resp. [5.3.2](#_bookmark24)) corre- sponds to the case where *l* labels a transition of Type 1 (resp. Types 2 and 3). The reader who is not interested by these details, may directly go to Sect. [5.4](#_bookmark29) which presents in a formal way the iterative process of SEA construction.

* 1. *Construction of states and transitions*

From Sect. [3.3](#_bookmark6), a transition TR of *B* is labeled: E if TR is of Type 1, *lT* if TR is of Type 2, and (E *, lT* ) if TR is of Type 3; where *lT* is in the form (*σ*) or (*σ,* S). Recall that if TR is of Type 2 or 3, then it has a corresponding transition in *A*' (and *A*). The index *T* in *lT* indicates that: *T* is the transition in *A*' (and *A*) corresponding to TR, and *lT* consists of *σ* and *Set* events (if any) that label *T* in *A*'. The general form of a label of a transition TR of SEA can be written (E *, lT* ) where: *lT* =if TR is of Type 1, and E = ∅ if TR is of Type 2.

Notations 5.3 *:*

1. *Let X be a system of linear equations and inequations using a set of variables x*1*, x*2*,* ··· *, xN . A nonnegative solution of X is any N-tuple* (*a*1*, a*2*,* ··· *, aN* ) *that satisﬁes X, where ai is a nonnegative value of xi. Let Sol*≥*0* (*X* ) *denote the set of nonnegative solutions of X.*
2. *If S* = *e*1*e*2 ··· *en is a sequence, then:*

|*S*| *denotes the* length *of S (i.e.,* |*S*| = *n),*

*S* [*i* ] *denotes the ith element of S (i.e., S* [*i* ] = *ei),*

*S* [/ *i*] *is obtained from S by* removing *the ith element of S.*

1. *Let ci and cj be two clocks, and* ∆*C*1 *and* ∆*C*2 *be two* ∆*Clock-Conds using ci* — *cj (see Def.* [*5.4*](#_bookmark16)*).* ∆*C1* H∆*C2 is obtained by* combining ∆*C*1 *and* ∆*C*2 *into a single* ∆*Clock-Cond that corresponds to the conjunction of* ∆*C*1 *and*

∆*C*2*. For example, if* ∆*C*1 = (*m*1 *< ci*—*cj < m*2) *and* ∆*C*2 = (*m < ci*—*cj*)*, then the result of* combination *is* max(*m*1*, m*) *< ci* — *cj < m*2*.*

1. *Let ci and θi be two clocks, and C be a Clock-Cond using ci. C* ⟨*ci* '→ *θi* — *ci*⟩ *is obtained from C by* replacing *ci by θi* — *ci.*
2. *Let* ∆*C be a set of* ∆*Clock-Conds, and θi be a clock possibly used in* ∆*C.*

∆*C* ⟨/ *θi*⟩*is obtained from* ∆*C by* eliminating *θi (if any) from* ∆*C by rewrit- ing its* ∆*Clock-Conds. For that purpose, every pair of* ∆*Clock-Conds in*

∆*C that use a θi, are added with each other in order to eliminate θi. The* *addition of two* ∆*Cr* (*θi , cj* ) *and* ∆*Cr* (*θi , ck* ) *for the elimination of θi is re- alized as follows: 1)* ∆*Cr* (*θi , cj* ) *is expressed using θi* — *cj; 2)* ∆*Cr* (*θi , ck* ) *is expressed using ck* — *θi; 3) the members of the two* ∆*Clock-Conds are added in order to obtain a* ∆*Clock-Cond expressed using ck* — *cj; 4) the obtained* ∆*Clock-Cond is combined with* ∆*Cr* (*cj , ck* )*, if any.*

*If θi is used in a single* ∆*Clock-Cond, the elimination consists of removing this* ∆*Clock-Cond.*

* + 1. *Study of transitions labeled* (S) *(i.e., of Type 1)*

Let *q* be a state of the SEA *B* = *StepTwo*(*A*') and S be a set of expirations.

Let *X* be the following system of inequations (formally described as a *set*

of inequations) where the variables are clocks:

*X* = ∆*Cq* ∪ {*ci* — *ki <* 0 : *Exp*(*ci* ; *ki* ) ∈ S}

∪ {*ci* — *ki* = *cj* — *kj* : *Exp*(*ci* ; *ki* )*, Exp*(*cj* ; *kj* ) ∈ S}

∪ {*ci* — *ki < cj* — *kj* : *Exp*(*ci* ; *ki* ) ∈ *EXP* (*q* ) \ S *, Exp*(*cj* ; *kj* ) ∈ S}

The following lemma can be used to determine whether S is enabled in *q*. It states that S is enabled in *q iff* : each *Exp*(*ci* ; *k* ) ∈ S is the expected next expiration of *ci*, and *X* has nonnegative solutions.

Lemma 5.1 [4](#_bookmark23) (*q* →E )! e (S ⊆ *EXP* (*q* ) *and Sol* (*X* ) /= ∅)*.*

≥*0*

When S is enabled in *q* (noted (*q* →E )!), let us express formally the procedure for computing the state *r*, noted (*q* →E ), reached from *q* after the occurrence of S:

4 Proof in Section [A.1](#_bookmark57)

1. *r* := *q*
2. ∀*Exp*(*ci* ; *k* ) ∈ S: |*Kr* (*ci* )| = 0 ⇒ *Cr* (*ci* ) := (*k < ci*);
3. |*Kr* (*ci* )| *>* 0 ⇒ *Cr* (*ci* ) := (*k < ci < Kr* (*ci* )[*1* ]),
4. *Kr* (*ci* )[/ 1].
5. ∀*Exp*(*ci* ; *ki* )*, Exp*(*cj* ; *kj* ) ∈ S: ∆*Cr* (*ci , cj* ) := (*ci* — *cj* = *ki* — *kj*).
6. ∀*Exp*(*ci* ; *ki* ) ∈ S *,* ∀*Exp*(*cj* ; *kj* ) ∈ *EXP* (*q* ) \ S:
7. ∆*Cr* (*ci , cj* ) := ∆*Cr* (*ci , cj* )H(*ki* — *kj < ci* — *cj* ).
8. ( ∀*ci* ∈ C*,* ∃*k >* 0*, Cr* (*ci* ) = (*k < ci*) ) ⇒ ∆*Cr* := ∅.

Let us give some explanations of the above computation of (*q* →E ):

*Line 1* : *r* is initialized with *q*.

*Lines 2-4* : For each *Exp*(*ci* ; *k* ) ∈ S, the aim is to update *Cq* (*ci* ) and *Kq* (*ci* ).

*Line 2* : We consider the case where an *Exp*(*ci* ; *k* ) ∈ S is the last expected expiration of *ci*. The assignment models the fact that *Exp*(*ci* ; *k* ) has occur- rred and that no other expiration of *ci* is foreseen.

*Lines 3-4* : We consider the case where: *Exp*(*ci* ; *k* ) is the expected next expiration of *ci*, and another *Exp*(*ci* ; *k* ') becomes expected after *Exp*(*ci* ; *k* ). In *Line 3* the assignment models the fact that *Exp*(*ci* ; *k* ) has occurred and that *Exp*(*ci* ; *k* ') is the expected new expiration. (*k*' is indicated by *Kr* (*ci* )[*1* ].) In *Line 4* , *k*' is removed from *Kr* (*ci* ) because *Kr* (*ci* ) is a se- quence that specifies the values to which *ci* will expire *after* its next expi- ration *Exp*(*ci* ; *k* ').

*Line 5* : Let *τ* be the time of occurrence of S. For every *Exp*(*ci* ; *ki* ) ∈ S, *ci* — *ki* = 0 at time *τ* . Therefore for every *Exp*(*ci* ; *ki* ) and *Exp*(*cj* ; *kj* ) ∈ S, *ci* — *ki* = *cj* — *kj* = 0 at time *τ* . Since all the clocks increase at the same rate, *ci* — *ki* = *cj* — *kj* (i.e., *ci* — *cj* = *ki* — *kj*) is also satisfied before and after *τ* . This equality replaces ∆*Cq* (*ci , cj* ). Intuitively, from the observation of the simultaneity of *Exp*(*ci* ; *ki* ) and *Exp*(*cj* ; *kj* ), we can replace ∆*Cq* (*ci , cj* ) by a more precise information.

*Lines 6-7* : Let *τ* be the time of occurrence of S. For every *Exp*(*ci* ; *ki* ) ∈ S, *ci* — *ki* = 0 at time *τ* . For every *Exp*(*cj* ; *kj* ) ∈ *EXP* (*q* ) \ S, *cj* — *kj <* 0 at time *τ* , because we are in the case where the expirations of *EXP* (*q* ) \S occur *after* S. This case is possible because S is enabled in *q*. Therefore *cj* — *kj < ci* — *ki* at time *τ* . Since all the clocks increase at the same rate, *cj* — *kj < ci* — *ki* (i.e., *ki* — *kj < ci* — *cj*) is also satisfied before and after *τ* . This ∆Clock-Cond is *combined* with ∆*Cq* (*ci , cj* ). For example, if

∆*Cq* (*ci , cj* ) is in the form *m*1 *< ci* — *cj < m*2, then the result of *combination*

is max(*m*1*, ki* — *kj*) *< ci* — *cj < m*2. Intuitively, from the observation that

*Exp*(*ci* ; *ki* ) is before *Exp*(*cj* ; *kj* ), we replace ∆*Cq* (*ci , cj* ) by a more precise

information.

*Line 8* : ∆*Cr* is emptied when all the clocks are inactive, because in this case ∆Clock-Conds are irrelevant (and thus, useless).

* + 1. *Study of transitions of types 2 and 3*

Let *q* be a state of the SEA *B* = *SetExp*(*A*) = *StepTwo*(*A*'), *lT* consist of *σ* and *Set* events (if any) that label a transition *T* in *A*', and S be a set of expiration(s).

The following lemma can be used to determine whether *lT* is enabled in *q*.

Lemma 5.2 [5](#_bookmark27) (*q* →*lT* )! iff*:*

* + - 1. *lT* ∈ *OUT* (*Lq* )*;*
      2. ∀“*<Exp*(*ci* ; *k* )” *or* “≤ *Exp*(*ci* ; *k* )” ∈ G*T ,* ∃*k*' ≤ *k such that Cq*(*ci* ) = (*u < ci < k*')*;*
      3. / ∃“= *Exp*(*ci* ; *k* )” ∈ G*T ;*
      4. ∀“*>Exp*(*ci* ; *k* )” or “≥ *Exp*(*ci* ; *k* )” ∈ G*T ,* ∃*k*' ≥ *k such that Cq*(*ci* ) = (*k*' *< ci < u*) or *Cq* (*ci* ) = (*k*' *< ci*)*.*

The following lemma can be used to determine whether (S *, lT* ) (i.e., label of a transition of Type 3) is enabled in *q*:

Lemma 5.3 [6](#_bookmark28) (*q* E *,lT* )! iff*:*

→

1. *lT* ∈ *OUT* (*Lq* )*;*
2. (*q* →E )!*;*
3. ∀“*<Exp*(*ci* ; *k* )” ∈ G*T : Exp*(*ci* ; *k* ) /∈ S *,* and

∃*k*' ≤ *k such that Cq*(*ci* ) = (*u < ci < k*')*;*

1. ∀“≤ *Exp*(*ci* ; *k* )” ∈ G*T ,* ∃*k*' ≤ *k such that Cq* (*ci* ) = (*u < ci < k*')*;*
2. ∀“= *Exp*(*ci* ; *k* )” ∈ G*T : Exp*(*ci* ; *k* ) ∈ S *,* and *Cq* (*ci* ) = (*u < ci < k*)*;*
3. ∀“*>Exp*(*ci* ; *k* )” ∈ G*T ,* ∃*k*' ≥ *k such that*

*Cq*(*ci* ) = (*k*' *< ci < u*) or *Cq* (*ci* ) = (*k*' *< ci*)*;*

1. ∀“≥ *Exp*(*ci* ; *k* )” ∈ G*T :*

∃*k*' ≥ *k such that Cq* (*ci* ) = (*k*' *< ci < u*) or *Cq*(*ci* ) = (*k*' *< ci*)*;* or

*Exp*(*ci* ; *k* ) ∈ S and *Cq*(*ci* ) = (*u < ci < k*)*.*

Let TR be a transition labeled (S *, lT* ) of Type 2 or 3 (S = ∅ if TR is

→

of Type 2). When (S *, lT*

) is enabled in *q* (noted (*q* (E *,lT* ))!), let us express

5 Proof in Section [A.2](#_bookmark58)

6 Proof in Section [A.3](#_bookmark59)

formally the procedure for computing the state *v*, denoted (*q* E *,lT* ), reached

→

from *q* after the execution of TR:

1. *v* := (*q* →E )

1→7

/\* Execution of *Lines 1-7* of the procedure of Sect. [5.3.1](#_bookmark21) \*/

1. *Lr* := destination location of *T*
2. 6*Set*(*ci* ; *k1 , k2 ,* ··· *, km* ) ∈ Z*T* (and thus, c*i* ∈ Z*T* ):

4. ∆*Cr* ⟨*ci* '→ *θi*⟩

1. / E*Exp*(*ci* ; *k* ) ∈ S ⇒ ∆*Cr* := ∆*Cr* ∪ {*Cr* (*ci* )⟨*ci* '→ *θi* — *ci*⟩}
2. E*Exp*(*ci* ; *k* ) ∈ S ⇒ ∆*Cr* := ∆*Cr* ∪ {*θi* — *ci* = *k*}
3. 6*cj* /∈ *ZT* (i.e., / E*Set* (*cj* ; ∗) ∈ Z*T* ):
4. / E*Exp*(*cj* ; *k* ) ∈ S ⇒ ∆*Cr* := ∆*Cr* ∪ {*Cr* (*cj* )⟨*cj* '→ *cj* — *ci*⟩}
5. E*Exp*(*cj* ; *k* ) ∈ S ⇒ ∆*Cr* := ∆*Cr* ∪ {*cj* — *ci* = *k*}
6. *Cr* (*ci* ) := (0 *< ci < k*1)
7. *Kr* (*ci* ) := *k*2 ··· *km*
8. 6*ci, cj* ∈ *ZT* : ∆*Cr* := ∆*Cr* ∪ {*cj* — *ci* = 0}
9. 6*θi*: ∆*Cr* ⟨/ *θi*⟩.

14. *v* := (*q* →E )

8

/\* Execution of *Line 8* of the procedure of Sect. [5.3.1](#_bookmark21) \*/

Let us give some explanations of the above computation of (*q* E *,lT* ).

→

*Lines* 1 and 14 : The simultaneity of S and *T* is conceptually equivalent to the execution of S which is *immediately* followed by *T* . We first determine the intermediate (virtual) state *v* reached from *q* after the execution of S. We compute *v* by using (*q* →E ) without its last *Line 8* . The latter is used at the end of (*q* E *,lT* ) (in *Line 14* ) because it is used only to finalize the computation of a “real” state. The aim of *Lines 1-13* is to determine the (real) state *v* reached from *v* after the execution of *T* . In (*q* E *,lT* ), *v* and *v* are indicated by the same variable *v*, because the intermediate state is directly modified for obtaining the final reached state. For the sake of clarity, in the following explanations, the two states are indicated differently by *v* and *v*.

→

→

*Lines 2-13* are executed only if *lT* /=, because when *lT* =, the transition is of Type 1 and the reached state (*q* →E ) is computed by Lines 1 and 14 (i.e., the procedure of Sect. [5.3.1](#_bookmark21)). The following explanations assume *lT* /=.

*Line 2* : The location of the reached state is the destination location of *T* .

*Lines 3-13* : If *T* resets no clock, then *T* has no influence on the values of clocks and State *v* is determined by Lines 1, 2 and 14. The aim of *Lines 3-13* is to compute the effect of the resets of *T* on *v*.

*Lines 3-11* : The aim is to compute the effect on *v*, of the reset of each clock.

*Line 4* : For every *ci* that is reset by *T* , we use a fictitious clock *θi* that is equal to *ci* (before the reset of *ci*) and that is not reset by *T* . Therefore, all the ∆Clock-Conds using *ci* that were satisfied before *T* , will continue to be satisfied after *T* if we replace *ci* by *θi*.

*Line 5* : Let *τ* be the time when *T* is executed and *ci* is reset. If *ci* does not expire at time *τ* , then *Cv* (*ci* ) is satisfied before the reset of *ci* and would be satisfied at instant *τ* if *ci* was not reset. At time *τ* , *θi* has the value *ci* would have if it was not reset, and *ci* is null. Therefore at time *τ* , *θi* — *ci* has the value *ci* would have if it was not reset. And since *θi* — *ci* is constant from *τ* and before another reset of *ci*, we deduce that in this period, *θi* — *ci* has the value *ci* would have at time *τ* if it was not reset. Therefore, *Cv* (*ci* ) is satisfied after *T* if we replace *ci* by *θi* — *ci*. This information is inserted into ∆*Cr* .

*Line 6* : Let *τ* be the time when *T* is executed. *θi* = *k* at time *τ* because *Exp*(*ci* ; *k* ) occurs at time *τ* , and *ci* = 0 at time *τ* because *ci* is reset at time *τ* . Therefore, *θi* — *ci* = *k* at time *τ* . Since *θi* — *ci* is constant from *τ* and before another reset of *ci*, we deduce that *θi* — *ci* = *k* in this period. This informa- tion is inserted into ∆*Cr* .

*Lines 7-9* are associated to a *ci* reset by *T* and consider every *cj* not reset by *T* .

*Line 8* : Let *τ* be the time when *T* is executed and *ci* is reset. If *cj* (*not* reset by *T* ) does not expire at *τ* , then *Cv* (*cj* ) is satisfied at time *τ* . At time *τ* , *cj* — *ci* has the value of *cj*, because *ci* is reset at time *τ* . Since *cj* — *ci* is constant from *τ* and before another reset of *ci* or *cj*, we deduce that in this period, *cj* — *ci* has the value that *cj* has at time *τ* .

Therefore, *Cv* (*cj* ) is satisfied after *T* if we replace *cj* by *cj* — *ci*. This information is inserted into ∆*Cr* .

*Line 9* : Let *τ* be the time when *T* is executed. *cj* = *k* at time *τ* because *Exp*(*cj* ; *k* ) occurs at time *τ* , and *ci* = 0 at time *τ* because *ci* is reset at time *τ* . Therefore *cj* — *ci* = *k* at time *τ* . Since *cj* — *ci* is constant from *τ* and before another reset of *ci* or *cj*, we deduce that *cj* — *ci* = *k* in this period. This information is inserted into ∆*Cr* .

*Lines 10-11* : We insert in *Cr* the Clock-Cond that specifies the status of *ci* just after its reset. If *Set*(*ci* ; *k1 ,* ··· *, km*) is the *Set* event of *ci* that is associated to *T* , then: in *Line 10* we insert the information that *ci* has been reset and that its next foreseen expiration is *Exp*(*ci* ; *k1* ), and in *Line*

*11* we insert the information which specifies the expirations of *ci* that are foreseen *after Exp*(*ci* ; *k1* ).

*Line 12* : we insert into ∆*Cr* the information that the clocks reset by *T* are

equal with each other after the execution of *T* .

*Line 13* : The use of *θi* is just a trick to deal with clocks that are reset while they are active. Now, we can eliminate every *θi* from ∆*Cr* . For that purpose, every pair of ∆Clock-Conds in ∆*Cr* that use a *θi*, are added with each other in order to eliminate *θi*. (See Item 7 in Notations [5.3](#_bookmark20) for more details about elimination of *θi*.)

* 1. *Construction of SEA*

The principle of the iterative procedure for computing *B* = *StepTwo*(*A*') (where *A*' = *StepOne*(*A*)) is explained in [5.2](#_bookmark18). The basis of this procedure can be formally expressed by the following fixpoint characterization:

Let us set the SEA *B*0 = (*Q*0*,* Γ0*, δ*0*, q*0) = ({*q*0}*,* ∅*,* ∅*, q*0), where *q*0 has been specified in Sect. [5.2](#_bookmark18). And let us consider the suite of SEA *Bi* = (*Qi,* Γ*i, δi, q*0), defined by *Bi*+1 = Ω(*Bi*), for *i* ≥ 0, where the operator Ω: SEA→ SEA is defined by:

*Q* = *Q* ∪ {(*q* →*l* ) : *q* ∈ *Q ,l* ∈ (*2 EXP*(*q*) × *OUT* (*L* )) \ {(∅*,* )}*,* (*q* →*l* )!}

*i*+1 *i i q*

*δ* = *δ* ∪ {*q* —→*l r* : *q* ∈ *Q ,* (*q* →*l* )!*,v* = (*q* →*l* ) ∈ *Q* }

*i*+1 *i*

*i i*+1

Γ*i*+1

= Γ ∪ {*l* : E(*q* —→*l*

*r* ) ∈ *δi*+1}

In the above expressions of *δ* and Γ , *q* —→*l*

*i*

*i*+1 *i*+1

state *q* to state *v* which is labeled *l*.

*r* denotes a transition from

Lemma 5.4 [7](#_bookmark33) Ω *has a ﬁxpoint that is obtained after a* finite *number of iter- ations. Formally:* E*p* ≥ 0 *s.t.* 6*n* ≥ *p: Bn* = *Bp.*

The SEA *B* = *StepTwo*(*A*') is indeed the fixpoint of Ω.

# Correctness and properties of *SetExp*

* 1. *Correctness*

In this section, we present a theorem which states correctness of *SetExp*.

Let *LSEA* denote the regular language of a SEA *B*. Each sequence of *LSEA*

*B B*

is in the form “*E*1*E*2 ··· *Ei* ·· ·”, where each *Ei* is the label of a transition of *B*,

and thus, consists of one event or several (simultaneous) events (see Sect. [3.3](#_bookmark6)).

Definition 6.1 *The* timed language of a SEA *B (TLSEA) is deﬁned as follows, where τi are real values:* (*E*1*, τ*1)(*E*2*, τ*2) ··· ∈ *TLSEA* iff*:*

*B*

*B*

7 Proof in Section [A.4](#_bookmark60)

*E*1*E*2 ··· ∈ *LSEA (see Def.* [*3.1*](#_bookmark7)*),* 0 *< τ*1 *< τ*2 *<* ·· ·*, and*

*B*

6*Ei, Ej, i < j:* If *Ei contains Set*(*c*; *k* )*, Ej contains Exp*(*c*; *k* )*, and*

*no Em (i < m < j) contains Set* (*c*; ∗)*,*

Then*: τj* = *τi* + *k.*

That is, *TLSEA* contains every timed sequence obtained from a sequence of

*B*

*SEA B*

*L*

by: associating a time to each event, and respecting the consistency

condition (i.e., each *Set*(*c*; *k* ) and corresponding *Exp*(*c*; *k* ) are separated by

the delay *k*).

Definition 6.2 *A TA A is said* equivalent *to a SEA B* iff *TLTA (see Def.* [*2.1*](#_bookmark4)*) is obtained from TLSEA by removing all the Set and Exp events.*

*A*

*B*

Intuitively, a TA *A* is equivalent to a SEA *B iff* the behaviors of *A* and *B* cannot be differentiated by an observer who does not see (or ignores) *Set* and *Exp* events.

Theorem 6.1 [8](#_bookmark36) *Every TA A is equivalent to its corresponding SEA SetExp*(*A*)*.*

Theorem [6.1](#_bookmark34) states correctness of *SetExp* and implies the possibility to transform a study of a system modelled by a TA *A* into a non-real-time form (i.e., *SetExp*(*A*)), and thus, we can adapt non-real-time methods of study. This idea has been used in conformance testing [[7](#_bookmark48),[10](#_bookmark51)] and supervisory con-

trol [[11](#_bookmark52),[8](#_bookmark49),[9](#_bookmark50)]. Therefore, this theorem confirms correctness of the methods in [[7](#_bookmark48),[10](#_bookmark51),[11](#_bookmark52),[8](#_bookmark49),[9](#_bookmark50)].

* 1. *Properties*

Property 6.1 *(General property) If in a TA, we multiply by the same value K all the constants used in the guards, the results of Steps 1 and 2 are not modiﬁed, modulo the multiplication by K of every constant used in a Set or Exp event.*

Property [6.1](#_bookmark35) can be intuitively justified as follows: multiplying all the con- stants of guards by the same value is equivalent to changing the unit of time, which has no influence on the generation of *Set* and *Exp* events. Here is a

mathematical justification: multiplying by the same value all the constants of guards, implies multiplying by the same value all the constants of the inequa- tions (Clock-Conds, ∆Clock-Conds) defining the states of SEA, which has no influence on the solutions of these inequations.

Property [6.1](#_bookmark35) is interesting because: (i) state space in the obtained SEAs does not increase when we multiply by the same value all the constants used in the transitions’ guards, and (ii) if *SetExp* has been applied to generate a SEA

8 Proof in Section [A.5](#_bookmark66)

*B* from a TA *A*, we can deduce straightforwardly every SEA corresponding to any TA *A*' that is similar to *A* modulo a multiplication of constant(s) used in guards.

We define now a more interesting property but that holds only for certain TA, for instance the TA of Fig. [1](#_bookmark3), and we consider a class of TAs for which this property holds.

Property 6.2 *(Property of a class of TA) We consider a class of TAs such that for every TA A of this class: let k*1*, k*2*,* ··· *, kp (where ki < ki*+1*, for i* = 1*,* 2*,* ··· *p* — 1*) be all the constants used in the guards of A.*

If *every ki is replaced by mi such that mi < mi*+1*, for i* = 1*,* 2*,* ··· *p* — 1*,* Then *the SEA SetExp*(*A*) *is not modiﬁed, modulo the replacement of every ki by mi.*

Property [6.2](#_bookmark37) holds for the TA of Figure [1](#_bookmark3), that is, if we replace the three values 1*,* 2*,* 3 used in the guards of the transitions, by any other integer values *k, m, n* respectively, such that *k < m < n*, the results of Figures [3](#_bookmark15) and [4](#_bookmark17) are not modified, modulo the replacement of 1*,* 2*,* 3 by *k, m, n* in *Set* and *Exp* events.

Note that we have studied several examples, and this property is satisfied in most of them.

Property [6.2](#_bookmark37) is more interesting than Property [6.1](#_bookmark35) because it does not restrict the modification of constants of guards by multiplying all of them by the *same* value. The determination of a class of TA in which Property [6.2](#_bookmark37) holds, is not trivial and we intend to study it in a near future.

Remark 6.1 *(About Property* [*6.2*](#_bookmark37)*)*

*In all the examples of TA we have studied, we note that, even for a TA where Property* [*6.2*](#_bookmark37) *does not hold, the state space of the corresponding SEA can change but its size does not increase signiﬁcantly with the magnitudes of constants used in guards.*

Properties [6.1](#_bookmark35) and [6.2](#_bookmark37), and Remark [6.1](#_bookmark38) show an advantage of using SEA instead of RA. In fact, contrary to RA, *in practice* the state space of SEA does not increase with the magnitudes of the constants used in timing constraints.

We have used the term “in practice” because Remark [6.1](#_bookmark38) holds for all the examples we have studied, but we have no guarantee that it holds for every TA. We intend to study this aspect in a near future.

Let us now give an upper bound of the number of states of a SEA *B* obtained from a TA *A* (i.e., *B* = *SetExp*(*A*)). Let *k* be the greatest constant used in timing constraints. For each clock *ci*, let us consider the distinct *Set* events associated to *ci*, and let *pi,j* be the nΣumber of constants in each of these *Set*

events, for *j* = 1*,* ·· ·. Let then *pi* = *j*=1*,*··· *pi,j* and *p* = max*i*=1*,*···*,Nc pi*. And

recall that |L| and |C| are the numbers of locations and clocks of *A*, respectively.

Lemma 6.1 [9](#_bookmark44) |L|*p*|C|2*k*|C|2 *is an upper bound of the number of states of B.*

Lemma [6.1](#_bookmark39) presents a coarse upper bound which is never reached. We intend in a near future to compute a more accurate upper bound.

# Conclusion

We propose a method *SetExp* that transforms a TA *A* into an equivalent SEA *B* which uses two types of events: *Set* and *Exp*, in addition to the events of the TA. *A* and *B* are *equivalent* , in the sense that they specify the same order and timing constraints. Applicability of *SetExp* has been demonstrated in [[7](#_bookmark48),[10](#_bookmark51),[11](#_bookmark52),[8](#_bookmark49),[9](#_bookmark50)], where *SetExp* is used as a black-box. In the present study, we show *how SetExp* can be realized.

In the near future, we intend to investigate the following issues:

* A software tool implementing *SetExp* has been recently realized. We intend to use this tool to implement a conformance testing method based on [[10](#_bookmark51)] and a supervisory control method based on [[9](#_bookmark50)].
* In the architectures proposed in [[7](#_bookmark48),[10](#_bookmark51),[11](#_bookmark52),[8](#_bookmark49),[9](#_bookmark50)], *SetExp* is applicable for *cen- tralized* systems. It would be interesting to design a transformation appli- cable for distributed systems.
* The state space of SEA increases exponentially with the number of clocks, and in practice this state space does not increase significantly with the magnitudes of the constants used in timing constraints. It is interesting to quantify more rigorously how the state space increases with the magnitudes

of constants. And it is also interesting to study more rigorously the com- plexity of *SetExp* in terms of memory and time that are necessary for the computation of SEA.

* Determination of a class of TA for which Property [6.2](#_bookmark37) is satisfied.
* For the sake of simplicity, we have considered only TA without invariants. We intend to develop an extension of *SetExp* supporting invariants.

# References

1. Alur, R., *Timed automata*, in: *Proc. 11th Intern. Conf. on Comp. Aided Verif. (CAV)* (1999),

pp. 8–22.

1. Alur, R., C. Courcoubetis and D. Dill, *Model checking for real-time systems*, in: *Proc. IEEE Sympos. on Logic in Computer Science*, 1990.
2. Alur, R., C. Courcoubetis, N. Halbwachs, D. Dill and H. Wong-Toi, *Minimization of timed transitions systems*, in: *CONCUR* (1992), pp. 340–354.

9 Proof in Section [A.5.1](#_bookmark67)

1. Alur, R. and D. Dill, *A theory of timed automata*, Theoretical Computer Science 126 (1994),

pp. 183–235.

1. Kang, I. and I. Lee, *State minimization for concurrent system analysis based on state space exploration*, in: *Proc. Conf. On Computer Assurance*, 1994, pp. 123–134.
2. Kang, I. and I. Lee, *An efficient state space generation for analysis of real-time systems*, in:

*Proc. Intern. Sympos. on Soft. Testing and Analysis (ISSTA’96)*, 1996.

1. Khoumsi, A., *A method for testing the conformance of real time systems*, in: *Proc. 7th Intern. Sympos. on Formal Techn. in Real-Time and Fault Toler. Systems (FTRTFT)*, Oldenburg, Germany, 2002,

[http://www.gel.usherb.ca/khoumsi/Research/Public/FTRTFT02.ps.](http://www.gel.usherb.ca/khoumsi/Research/Public/FTRTFT02.ps)

1. Khoumsi, A., *Supervisory control of dense real-time discrete-event systems with partial observation*, in: *Proc. 6th Intern. Workshop on Discrete Event Systems (WODES)*, Zaragoza, Spain, 2002,

[http://www.gel.usherb.ca/khoumsi/Research/Public/WODES02.ps.](http://www.gel.usherb.ca/khoumsi/Research/Public/WODES02.ps)

1. Khoumsi, A., *Supervisory control for the conformance of real-time discrete-event systems*, in: *Proc. 7th Intern. Workshop on Discrete Event Systems (WODES)*, Reims, France, 2004, [http://www.gel.usherb.ca/khoumsi/Research/Public/WODES04.ps.](http://www.gel.usherb.ca/khoumsi/Research/Public/WODES04.ps)
2. Khoumsi, A., T. J´eron and H. Marchand, *Test cases generation for nondeterministic real-time systems*, in: *Proc. Formal Approaches to TEsting of Software (FATES), LNCS 2931* (2003), [http://www.gel.usherb.ca/khoumsi/Research/Public/FATES03.ps.](http://www.gel.usherb.ca/khoumsi/Research/Public/FATES03.ps)
3. Khoumsi, A. and M. Nourelfath, *An efficient method for the supervisory control of dense real- time discrete event systems*, in: *Proc. 8th Intern. Conf. on Real-Time Computing Systems* *(RTCSA)*, Tokyo, Japan, 2002, [http://www.gel.usherb.ca/khoumsi/Research/Public/RTCSA02-Cont.ps.](http://www.gel.usherb.ca/khoumsi/Research/Public/RTCSA02-Cont.ps)
4. Ramadge, P. and W. Wonham, *The control of discrete event systems*, Proc. IEEE 77 (1989),

pp. 81–98.

1. Tretmans, J., *Test generation with inputs, outputs and repetitive quiescence*, Software-Concepts and Tools 17 (1996).
2. Tripakis, S. and S. Yovine, *Analysis of timed systems based on time-abstracting bisimulations*, in: *Proc. 8th Intern. Conf. on Comp. Aided Verif. (CAV)* (1995), pp. 229–242.
3. Yannakakis, M. and D. Lee, *An efficient algorithm for minimizing real-time transition systems*, in: *Proc. 5th Conf. on Comp. Aided Verif. (CAV)* (1993), pp. 210–224.

# Proofs

In the following, by “is satisfied”, we mean “evaluates to *True*”.

* 1. *Proof of Lemma* [*5.1*](#_bookmark22)

Let us define :

*X*1 = {*ci* — *ki <* 0 : *Exp*(*ci* ; *ki* ) ∈ S},

*X*2 = {*ci* — *ki* = *cj* — *kj* : *Exp*(*ci* ; *ki* )*, Exp*(*cj* ; *kj* ) ∈ S},

*X*3 = {*ci* — *ki < cj* — *kj* : *Exp*(*ci* ; *ki* ) ∈ *EXP* (*q* ) \ S *, Exp*(*cj* ; *kj* ) ∈ S}. We have therefore : *X* = ∆*Cq* ∪ *X*1 ∪ *X*2 ∪ *X*3.

1. Since all the clocks progress at the same rate, the system *X*1 ∪ *X*2 ∪ *X*3 has positive solutions *iff* with the passing of time and if no clock is reset, the following items 2 and 3 will be satisfied simultaneously.
2. 6*Exp*(*ci* ; *ki* ) ∈ S : *ci* — *ki* = 0.
3. 6*Exp*(*cj* ; *kj* ) ∈ *EXP* (*q* ) \S : *cj* — *kj <* 0.
4. Item 2 holds *when and only when* the expirations of S occur simultane- ously.
5. Item 3 holds *while and only while* no expiration of *EXP* (*q* ) \S has oc- curred.
6. Items 1, 4 and 5 mean that the system *X*1 ∪*X*2 ∪*X*3 has positive solutions *iff* at the same future instant of time and if no clock is reset: *all* the expirations of S occur and *no* expiration of *EXP* (*q* ) \S occurs.
7. Item 6 implies that the system *X* = ∆*Cq* ∪ (*X*1 ∪ *X*2 ∪ *X*3) has positive solutions *iff* ∆*Cq* allows that at a same future time : *all* expirations of S occur and *no* expiration of *EXP* (*q* ) \S occurs.
   1. *Proof of Lemma* [*5.2*](#_bookmark25)
8. Item 3 of Lemma [5.2](#_bookmark25) means that every guard of *T* is in one of the following forms : “*<Exp*(*ci*; *k* )”, “≤ *Exp*(*ci* ; *k* )”, “*>Exp*(*ci* ; *k* )”, “≥ *Exp*(*ci* ; *k* )”.
9. Items 2 and 4 of Lemma [5.2](#_bookmark25) mean that every guard of *T* is satisfied in *q*

if it is in one of the following forms :

“*<Exp*(*ci*; *k* )”, “≤ *Exp*(*ci* ; *k* )”, “*>Exp*(*ci*; *k* )”, “≥ *Exp*(*ci* ; *k* )”.

1. The above Items 1 and 2 mean that every guard of *T* is satisfied in *q*.
2. Item 1 of Lemma [5.2](#_bookmark25) and the above Items 1 and 3 mean that *T* can be executed at any time in *q*, without being simultaneous to any expiration.
   1. *Proof of Lemma* [*5.3*](#_bookmark26)
3. In Item 3 of Lemma [5.3](#_bookmark26) :

“*Cq*(*ci* ) = (*u < ci < k*'), for *k*' ≤ *k*” means that every guard “*<Exp*(*ci*; *k* )” of *T* is satisfied in *q*;

“*Exp*(*ci*; *k* ) /∈ S” and Item 2 of Lemma [5.3](#_bookmark26) implies that such a guard “*<Exp*(*ci*; *k* )” remains satisfied at the occurrence of S.

1. In Item 4 of Lemma [5.3](#_bookmark26) :
   1. “*Cq*(*ci* ) = (*u < ci < k*'), for *k*' ≤ *k*” implies that every guard “≤ *Exp*(*ci* ; *k* )” of *T* is satisfied in *q*;
   2. “*Cq*(*ci* ) = (*u < ci < k*'), for *k*' ≤ *k*” and Item 2 of Lemma [5.3](#_bookmark26) imply that S occurs before or contains *Exp*(*ci* ; *k* ').

The above a) and b) imply that every guard “≤ *Exp*(*ci* ; *k* )” of *T* is and

*remains* satisfied at the occurrence of S.

1. Item 5 of Lemma [5.3](#_bookmark26) means that with the passing of time, every guard “= *Exp*(*ci*; *k* )” of *T* will be satisfied when and only when *ci* = *k*, i.e., simultaneously to *Exp*(*ci* ; *k* ) ∈ S.
2. Item 6 of Lemma [5.3](#_bookmark26) means that every guard “*>Exp*(*ci* ; *k* )” of *T* is satisfied in *q*. This guard remains satisfied at the occurrence of any expiration, and thus, at the occurrence of S.
3. In Item 7 of Lemma [5.3](#_bookmark26), for every guard “≥ *Exp*(*ci* ; *k* )” of *T* :
   1. “*Cq* (*ci* ) = (*k*' *< ci < u*) or *Cq* (*ci* ) = (*k*' *< ci*), for *k* ≤ *k*'” implies that “≥ *Exp*(*ci* ; *k* )” is and remains satisfied at the occurrence of S.
   2. “*Exp*(*ci* ; *k* ) ∈ S and *Cq*(*ci* ) = (*u < ci < k*)” implies that “≥ *Exp*(*ci* ; *k* )” will be satisfied when and only when *ci* = *k*, i.e., simultaneously to *Exp*(*ci* ; *k* ) ∈ S.

Item 7 of Lemma [5.3](#_bookmark26) means a) or b) holds.

1. In the above Items 1, 2, 3, 4, 5a, and 5b, all the guards of *T* are consid- ered.
2. The above items 1, 2, 4, and 5a correspond to the guards of *T* that are satisfied in *q*. These guards remain satisfied at the occurrence of S.
3. The above items 3 and 5b correspond to the guards of *T* that will be satisfied with the passing of time. Each of these guards will be satisfied simultaneously to an *Exp* event of S.
4. Item 2 of Lemma [5.3](#_bookmark26) and the above Item 8 mean that the guards of *T* that will be satisfied with the passing of time, can all be satisfied at the same time, simultaneously to S.
5. The above items 6, 7 and 8 mean that there is no non-satisfied guard of

*T* that will not be satisfied with the passing of time.

1. The above items 7, 9, and 10 and Item 1 of Lemma [5.3](#_bookmark26) mean that the simultaneous executions of *T* and S is possible in *q*.
   1. *Proof of Lemma* [*5.4*](#_bookmark30)

We have defined the suite *Bi*+1 = Ω(*Bi*), for *i* ≥ 0. Let *Qi,* Γ*i, δi* be such that

*Bi* = (*Qi,* Γ*i, δi, q*0). We have, by definition of the operator Ω (see Sect. [5.4](#_bookmark29)):

*Qi*+1 = *Qi* ∪ ∆*Qi*, *δi*+1 = *δi* ∪ ∆*δi*, Γ*i*+1 = Γ*i* ∪ ∆Γ*i*, where :

∆*Q* = {(*q* →*l* ) : *q* ∈ *Q ,l* ∈ (*2 EXP*(*q*)×(*OUT* (*L* ) ∪ {}))\{(∅*,* )}*,* (*q* →*l* )!}

*i i q*

∆*δ* = {*q* —→*l r* : *q* ∈ *Q ,* (*q* →*l* )!*,v* = (*q* →*l* ) ∈ *Q* }

*i*

*i i*+1

∆Γ = Γ ∪ {*l* : E(*q* —→*l*

*i*

*r* ) ∈ *δi*+1}.

We will also use the notation |*U*| for the *cardinal* (i.e., number of elements) of

a set *U* . In order to prove Lemma [5.4](#_bookmark30), let us prove the following five lemmas :

Lemma A.1 6*i* ≥ 0 : *Qi* ⊆ *Qi*+1*.*

Lemma A.2 6*p* ≥ 0 : ((*Qp* = *Qp*+1) ⇒ (*Bp*+2 = *Bp*+1))*.*

Lemma A.3 6*p* ≥ 0 : ((*Qp* = *Qp*+1) ⇒ (6*n > p* : *Bn* = *Bp*+1))*.*

Lemma A.4 E*K ﬁnite such that* 6*i* ≥ 0 : |*Qi*|≤ *K.*

Lemma A.5 |lim*i*→∞ *Qi*| *exists and is ﬁnite.*

* + 1. *Proof of Lemma* [*A.1*](#_bookmark61)

From the fact that 6*i* ≥ 0 : *Qi*+1 = *Qi* ∪ ∆*Qi*, we deduce straightforwardly that *Qi* ⊆ *Qi*+1.

* + 1. *Proof of Lemma* [*A.2*](#_bookmark62)
       1. By definition : *Qp*+1 = *Qp* ∪ ∆*Qp* and *Qp*+2 = *Qp*+1 ∪ ∆*Qp*+1.
       2. By definition :

∆*Q* = {(*q* →*l* ) : *q* ∈ *Q ,l* ∈ (*2 EXP*(*q*)×(*OUT* (*L* )∪{}))\{(∅*,* )}*,* (*q* →*l* )!},

*p p q*

∆*Q* ={(*q* →*l* ) : *q* ∈ *Q ,l* ∈(*2 EXP*(*q*)×(*OUT* (*L* )∪{}))\{(∅*,* )}*,* (*q* →*l* )!},

*p*+1 *p*+1 *q*

* + - 1. Item 2 implies that if *Qp* = *Qp*+1, then ∆*Qp* = ∆*Qp*+1.
      2. Items 1 and 3 imply that if *Qp* = *Qp*+1, then *Qp*+2 = *Qp*+1 = *Qp*.
      3. By definition : *δp*+1 = *δp* ∪ ∆*δp* and *δp*+2 = *δp*+1 ∪ ∆*δp*+1.
      4. By definition : ∆*δ* = {*q* —→*l r* : *q* ∈ *Q ,* (*q* →*l* )!*,v* = (*q* →*l* ) ∈ *Q* }, and

*p*

*p p*+1

∆*δ* = {*q* —→*l r* : *q* ∈ *Q ,* (*q* →*l* )!*,v* = (*q* →*l* ) ∈ *Q* }.

*p*+1

*p*+1 *p*+2

* + - 1. Items 4 and 6 imply that if *Qp* = *Qp*+1, then ∆*δp* = ∆*δp*+1.
      2. Items 5 and 7 imply that if *Qp* = *Qp*+1, then *δp*+2 = *δp*+1.
      3. By definition : Γ*p*+1 = Γ*p* ∪ ∆Γ*p* and Γ*p*+2 = Γ*p*+1 ∪ ∆Γ*p*+1.
      4. By definition : ∆Γ = Γ ∪ {*l* : E(*q* —→*l*

*p*

∆Γ*p*+1 = Γ ∪ {*l* : E(*q* —→ *r* ) ∈ *δp*+2}.

*l*

*r* ) ∈ *δp*+1}, and

* + - 1. Items 8 and 10 imply that if *Qp* = *Qp*+1, then ∆Γ*p* = ∆Γ*p*+1.
      2. Items 9 and 11 imply that if *Qp* = *Qp*+1, then Γ*p*+2 = Γ*p*+1.
      3. Items 4, 8 and 12 imply that if *Qp* = *Qp*+1, then : *Bp*+2 = *Bp*+1.
    1. *Proof of Lemma* [*A.3*](#_bookmark63)
       1. Let us consider a *p* ≥ 0 such that *Qp* = *Qp*+1. Lemma [A.2](#_bookmark62) implies that

*Bp*+2 = *Bp*+1.

* + - 1. Let us us consider a *n* ≥ *p* + 2 such that *Bn* = *Bn*−1. Therefore, *Qn* =

*Qn*−1.

From Lemma [A.2](#_bookmark62), we deduce that *Bn*+1 = *Bn*.

* + - 1. Items 1 and 2 imply by induction that :

if (*Qp* = *Qp*+1) for a given *p* ≥ 0, then 6*n* ≥ *p* +2 : *Bn* = *Bn*−1.

* + - 1. Item 3 is equivalent to :

if *Qp* = *Qp*+1 for a given *p* ≥ 0, then 6*n* ≥ *p* +2 : *Bn* = *Bp*+1.

* + - 1. Since *Bn* = *Bp*+1 for *n* = *p* + 1, Item 4 can be written :

if *Qp* = *Qp*+1 for a given *p* ≥ 0, then 6*n* ≥ *p* +1 : *Bn* = *Bp*+1.

* + 1. *Proof of Lemma* [*A.4*](#_bookmark64)

Let *A* = (L*,* A*,* C*,* T *, l*0) be the TA from which we construct *B*0*, B*1*, B*2*,* ·· ·. Recall that each state of SEA, and thus of *Bi*, is defined by three parts :

Part 1 : a location of L,

Part 2 : a Clock-Cond and an ExpSeq for each clock of C, Part 3 : zero or one ∆Clock-Cond for each pair of clocks of C.

* + - 1. The cardinal of Part 1 has a *ﬁnite* upper bound equal to |L|.
      2. The (integer) values to which a clock *ci* ∈ C is compared (i.e., the values used in guards using *ci*) are finite and their number is finite.
      3. Item 2 implies that the number of pairs (Clock-Cond, ExpSeq) using a clock *ci* ∈ C has a *ﬁnite* upper bound which we denote *U* .

For each clock *ci*, let us consider the distinct *Set* events associated to *ci*, and let *ki, j* beΣthe number of constants in each of these *Set* , for

*j* = 1*,* ·· ·. Let *pi* = *j*=1*,*··· *ki,j* and *p* = max*i*=1*,*···*,Nc pi*. It can be easily

proved that *U* can be taken equal to *p*.

* + - 1. Item 3 and the fact that the number of clocks of *A* is finite (= |C|) imply that *p*|C| is a (finite) upper bound of the cardinal of Part 2.
      2. For each pair of clocks *ci, cj* ∈ C, if *Mi* (resp. *Mj*) is the greatest value to which *ci* (resp. *cj*) is compared in the guards of *A*, then every ∆Clock- Cond using *ci* — *cj* is expressed by using (integer) values that fall within the interval [—*Mj, Mi*].
      3. Items 2 and 5 imply that for each pair of clocks *ci, cj* ∈ C, the number of possible ∆Clock-Conds using *ci* — *cj* has a *ﬁnite* upper bound which we denote *V* .

Since the width of interval [—*Mj, Mi*] is ≤ 2*k*, it can be easily proved that *V* can be taken equal to 22*k*, where *k* is the greatest constant used in timing constraints.

* + - 1. Item 6 and the fact that the number of pairs of clocks of *A* is finite (*<* |C|2 )

2

imply that 22*k* |C|

2

2

(= 2*k*|C|2 ) is a (finite) upper bound of the cardinal of

Part 3.

* + - 1. Items 1, 4 and 7 imply that |L|*p*|C|2*k*|C|2 is a (finite) upper bound of the number of states of every *Bi*.
    1. *Proof of Lemma* [*A.5*](#_bookmark65)

Lemma [A.1](#_bookmark61) implies that |lim*i*→∞ *Qi*| exists. And from Lemma [A.4](#_bookmark64), we deduce that that this |lim*i*→∞ *Qi*| is finite.

* + 1. *Proof of Lemma* [*5.4*](#_bookmark30)
       1. Lemmas [A.1](#_bookmark61) and [A.5](#_bookmark65) imply : E*i* ≥ 0 such that *Qi* = *Qi*+1
       2. Item 1 and Lemma [A.2](#_bookmark62) imply E*p* ≥ 0 such that 6*n > p* : *Bn* = *Bp*+1.
  1. *Proof sketch of Theorem* [*6.1*](#_bookmark34)

Let *A* be a TA, *A*' = *StepOne*(*A*) be obtained from *A* by applying Step 1, and

*B* = *StepTwo*(*A*') be the SEA obtained by applying Step 2 to *A*'.

1. Step 1 of *SetExp* relabels transitions of the TA *A* without changing its structure. The fact to reset a clock *c* in a transition Tr1, in order to com- pare further in a transition Tr2 the value of *c* with *k*, is clearly *equivalent* to :

resetting *c* and programming it such that it expires after a delay *k* (Sub- step 1a), and then checking in Tr2 whether the expiration of *c* has oc- curred (Substep 1b). Therefore, *A* and *A*' describe exactly the same order and timing constraints of the events other than *Set* and *Exp* events.

1. *SetExp* is realized by a fix-point method that constructs iteratively all the reachable states and the transitions of the SEA. This fix-point method converges after a finite number of iterations (Lemma [5.4](#_bookmark30)) and necessitates in each iteration :
   1. to determine the labels enabled in every state, and
   2. to construct the states reached by these enabled transitions. Therefore, this fix-point method generates a correct result *iff* (a) and (b) are realized correctly.
2. Lemmas [5.1](#_bookmark22), [5.2](#_bookmark25) and [5.3](#_bookmark26) imply that the labels enabled in every state are determined correctly.
3. The detailed explanations of the procedures in Sects. [5.3.1](#_bookmark21) and [5.3.2](#_bookmark24) can be considered as a proof sketch that the states reached by the enabled labels are constructed correctly.
4. Items 2, 3 and 4 imply that the fixpoint method is correct, i.e., Step 2 of *SetExp* is correct. Therefore, *B* accepts all and only the possible order of events that respects the specification of *A*'.
5. Item 5 and the fact that *TLSEA* (see Def. [6.1](#_bookmark32)) models the behaviour of *B*, mean that *TLSEA* contains all and only the possible order of events that respects the specification of *A*'.

*B*

*B*

1. Items 1 and 6 mean that *TLTA* (Def. [2.1](#_bookmark4)) is obtained from *TLSEA* by

*A*

*B*

removing all *Set* and *Exp* events, i.e., *A* and *B* are equivalent.

* + 1. *Proof of Lemma* [*6.1*](#_bookmark39)

See Proof of Lemma [A.4](#_bookmark64).