Development of an ML-based Verification Tool for Timed CSP Processes

Takeshi YAMAKAWA, Tsuneki OHASHI and Chikara FUKUNAGA [1]

Tokyo Metropolitan University, 1-1 Minami-Osawa, Hachoiji, Tokyo, 192-0397, Japan

Abstract. We report the development of a verification tool for Timed CSP processes. The tool has been built on the functional programming language ML. The tool interprets processes described in both timed and untimed CSP, converting them to ML functions, and executing those functions for the verification of refinement in the timed traces and timewise traces models. Using the programmability of higher order functionality, the description of CSP processes with ML has been synthesised naturally. The effectiveness of the tool is demonstrated with an example analysing implementations of Fischer's algorithm for the exclusive control of a shared resource in a multiprocessor environment.

Keywords. CSP, Timed CSP, formal methods, real-time,

Introduction

The model checking tool FDR2 [1] has been used widely for the formal verification of processes described in CSP theory [2] (hereafter called CSP or, sometimes, "untimed" CSP to emphasise CSP without time). CSP has been known to be suitable for the description of concurrent systems in which processes run in parallel and where synchronisation is achieved with shared events. CSP describes the transition of a process from one state to another with the occurrence of the events, but includes no concept for the timing of the occurrence of a particular event, the duration of an event or the time difference between consecutive events. For the formal verification of embedded or real time systems, an extended language of CSP that can express the timing of events or the start/stop of processes has been desired for long time. For this purpose, Timed CSP has been proposed [3] to include the concept of the timing in CSP. SystemCSP has been published later [4] by a group at the University of Twente. SystemCSP has been aimed as a design methodology for real time systems. With the development of such CSP extensions, it has been naturally expected to have tools for the formal verification of real time systems in the framework of the extended CSP. In 2007, a verification tool for Timed CSP, called PAT (Process Analysis Tool) [5], was released by J.Sun et al. (NU Singapore).

We have also developed a verification tool based on Timed CSP, called "Timed CSP Explorer". In this paper, the contents and basic principles of this tool are discussed in detail. The entire tool is written using the ML programing language. ML is used not only for the expression of Timed CSP processes as ML functions, but also for the lexical analysis, parser and verification stages of the tool. ML is adequate to express Timed CSP processes through higher order functions and recursion, since ML is a functional programing language. Custom data types, rather than simple types of integer, real and string, can be implemented with ML. This ability to describe a structurally complicated type is suitable to express objects like events or processes that have wide diversity of meaning in CSP. The concept of abstract data

[1] Corresponding Author: *Chikara Fukunaga*, E-mail: fukunaga@tmu.ac.jp.

typing, rather than the strong data typing, is also a benefit of ML for the flexible expression of CSP processes using polymorphic type checking.

ML has been originally developed as a theorem prover [6] . If one uses higher order functional programing appropriately, then inference rules and proof methods will be represented as functions. This technique can be applied also to verification tools for formal methods. In fact, the previous version (FDR1) of FDR2 has been built upon the Standard ML of New Jersey [7].

The current Timed CSP Explorer can verify refinements based on the timed traces and timewise traces models. We discuss the effectiveness of this tool in section 3 with an example of an algorithm for the exclusive control of a shared resource (critical region) where only one process can be accepted among several processes running concurrently.

1. Timed CSP

Timed CSPs [3] has been developed from CSP to include descriptions for the timed evolution of processes and the timing of the event occurrences in a CSP process. Timing information has been added quite naturally into the original CSP. Several new concepts and operators in Timed CSP are introduced briefly in this section.

1.1. Timed CSP Operators

In Timed CSP, time is modeled as non-negative real number and measured commonly in a single global clock for all the processes involved. A process P can be evolved to P' with time which is expressed as $P \overset{d}{\rightsquigarrow} P'$ (evolution transition) in which d is an arbitrary time unit ($d \geqslant 0$) in addition to a labeled transition with an event $Q \xrightarrow{\mu} Q'$ defined in CSP theory.

The timed event prefix operator gives a time value for its occurrence:

$$P = a@u \rightarrow P$$

The identifier, u, records the time from the start of P to the occurrence of the event a. An evolution step of a process to send a measured time between the events of $start$ and $finish$ can be expressed with the timed event prefix and the concept of the evolution transition together with the conventional labeled transition as:

$$\begin{aligned} P = (&start \rightarrow finish@u \rightarrow out!u \rightarrow SKIP) \xrightarrow{start} \\ &(finish@u \rightarrow out!u \rightarrow SKIP) \overset{3}{\rightsquigarrow} \\ &(finish@u \rightarrow out!(u+3) \rightarrow SKIP) \overset{2}{\rightsquigarrow} \\ &(finish@u \rightarrow out!((u+3)+2) \rightarrow SKIP) \xrightarrow{finish} \\ &(out!((0+2)+3) \rightarrow SKIP) \xrightarrow{out.5} SKIP \end{aligned}$$

The timeout operator is introduced as a choice with an event and the elapsed time from the start of the process. The expression:

$$(a \rightarrow P) \overset{d}{\rhd} Q$$

specifies that the process P will follow after the event a if a takes place within the time unit of d, otherwise the process Q will start after the timeout instead of P. With this operator a following process can be introduced as an operator:

$$WAIT\ d = STOP \overset{d}{\triangleright} SKIP$$

This process does nothing until time d. A delay of the start of a process for time d can be implemented with this operator as $WAIT\ d\,;P$. A new operator using this timeout can be also introduced, and is frequently used in the following sections. The operator is called the timed prefix and denoted as:

$$a \xrightarrow{d} P = a \rightarrow (STOP \overset{d}{\triangleright} P)$$

This example means that the start of P will be blocked for d time after the occurrence of the event "a". Minimum and maximum delays can be expressed using the timed prefix and the timeout respectively. The following process can be interpreted as a minimum delay since the event b will not take place at least d_1 after the event a:

$$C_1 = a \xrightarrow{d_1} (b \rightarrow SKIP)$$

The timeout operator indicates a maximum delay that a process guarantees to wait for its environment to engage with it:

$$C_2 = a \rightarrow ((b \rightarrow SKIP) \overset{d_2}{\triangleright} STOP)$$

since this can be interpreted that the process C_2 may stop if the event b has not occurred d_2 after the event a, although the actual occurrence of the event b depends on the environment.

The timed interrupt is denoted as:

$$P \triangle_d Q$$

This means that unless the process P is finished until d, it may be terminated and the process Q will follow afterwards; but if P is finished within d, then Q will not be invoked to start.

Not only have the new operators have been introduced in Timed CSP, but the semantics of existing CSP operators in the timed contexts must be also updated in Timed CSP if necessary. Modification of the CSP operators has been achieved just to add a few more deduction rules concerning the evolution transition in their definitions from the operational semantics point of view.

1.2. Refinement of a Process

Since a process described in CSP usually leaves several observable behaviours through the labeled transitions, the process can be also represented by the set of these behaviours. The phrase that the process Q refines the process P (denoted as $P \sqsubseteq Q$) for a particular behaviour means the set of behaviours of Q is a subset of the one for P. Assuming that the CSP description of P is the specification of a process and Q is an actual implementation, the implementation can be verified by establishing the refinement: $P \sqsubseteq Q$.

CSP usually uses three observable behaviours. These are traces, failures and divergences. A trace is sequences of events that a process makes for communication with other ones. A set of all the possible traces of a process P is denoted as $traces(P)$. A failure is represented with a set of two characteristics (s, X); $s \in traces(P)$ is a trace, and X is a set of events which can be refused after the trace s. We denote the set of failures of P as $failures(P)$. A divergence is a trace of P that will produce infinite sequence of unobservable events after that. Unobservable events are ones that are hidden in the process so that any external processes or environments can not observe the occurrence of the events. A set of divergences of P is written as $divergences(P)$.

Several models have been established to express the semantics of a process using the above observable behaviours. Verification of processes is made usually in the frameworks of these models. FDR2, for example, uses the following. The *traces* model uses only $traces(P)$ to describe a process: verification with this model checks the safety of P (i.e. that it will not do unspecified things). The *stable failures* model uses both $traces(P)$ and $failures(P)$ to represent P: verification with this model can check liveness properties (i.e. that it will do specified things) and the absence of deadlock. The *failures/divergences* model represents the process P with extended failures, denoted as $failures_{\perp}(P)$, and $divergences(P)$: verification here additionally checks the risk of P getting into a livelock state. The $failures_{\perp}(P)$ is defined as $failures(P) \cup (s, X)$, where $s \in divergences(P)$ and X is *any* set of events (to be refused after the trace s just like those defined for the failures model).

In Timed CSP, two more observable behaviours have been used in addition to the three behaviours in untimed CSP. These are timed traces and timed refusals. A timed trace is a sequence of timed events during the execution of a process. A timed event is represented as $(time, event)$.

A timed refusal is a record of timed events that were not possible to issue or accept for a state reached at some particular time. If a timed event (t_0, ev_0) is found in a set of timed refusals at a particular moment, the event ev_0 will not be possible in a state that the process can reach at t_0. Contrary to the untimed case, elements (timed events) contained in the set of timed refusals will not be fixed but varied during execution.

For the refinement of processes described in Timed CSP, several models have been proposed [3,8]. Among them, the timed traces model and the timed failures model are used for the refinement of Timed CSP processes. The timed traces model uses a set of timed traces only to describe a process behaviour, while the timed failures model uses both sets of the timed traces and timed refusals. Refinement using the timed failures model for timing sensitive processes will enable more discriminating verifications than one using the timed traces model. Correctness of the description of a process will be validated more reliably using the time information in a real time environment, even if the verification is achieved solely with the timed traces refinement.

2. Timed CSP Explorer

We have developed the "Timed CSP Explorer" tool for the refinement of processes described in Timed CSP. The refinement model for this application is based on the timed traces, enabling timed trace and trace timewise refinements to be verified. Trace timewise refinement verifies a process with the traces model in untimed CSP domain by ignoring the time information in a timed trace.

Timed CSP Explorer consists of two parts; the one is the ML code generator and the other one is the verifier. The ML code generator converts codes written in CSP to ML for both the specification and the process to be verified while the verifier constructs two sets of timed traces of the specification and the process (implementation) by executing actually the ML functions, and does the verification by checking if the set formed by the process is included (subset) in the one of the specification. In the following subsections, we discuss the mechanisms implemented in these two parts.

2.1. Describing Processes in ML

We have two strategies to convert the processes described in Timed CSP to ML. The first is to express a CSP operator as an ML higher order function which takes one or any number of processes as the input and return a function, and the second one is to install new ML data types for events as well as processes.

A CSP operator makes normally a new process as a result of the operation with several processes. If we write a CSP process as an ML function, then the operator must work to produce a new function from several input functions. With the technique of the higher order function embedded in ML, an entire logic of a CSP operator can be expressed throughout in ML. Although events and processes in CSP have wide diversity of the expression, each of these objects has a unique functionality in CSP. If these objects are implemented with new data types, original functionalities of these objects will be kept clear even in ML description.

2.1.1. New Datatypes Introduced

New data types can be introduced in an ML program using the `datatype` declaration. In Timed CSP Explorer, we have implemented three main datatypes for the ML representation of CSP processes. These are for processes, events and a chantype which specifies the type of a channel in an event.

The type `process` is declared as follows:

```
datatype process
  = Proc of (event -> process)
    | Stop
    | Skip
    | Bleep ;
```

In this `datatype` declaration, five items are declared, namely the type `process` and its four constructors. `Proc` is a function to take an event type variable as the argument and to return a process type value (one of the items listed above). The other constructors are aliases of type `process`. `Bleep` is a process that returns an error if a process receives a non-executable event.

The type `event` is defined as:

```
datatype event
  = Event of string*chanType ;
```

Events in CSP are classified into two types: those without any parameters (interaction type), and those with parameters (channel communication). However, we define just one type for `event`: the composition of a name (string) and parameters (numbers and/or strings). In ML, there is a facility called a *tuple* to store data of different types in one unit. The constructor `Event` of the type `event` shown above is such a tuple, which consists of a *string* and a *chanType*.

The type *chanType* is listed below. Among six kinds of constructors listed in this type, the last one (`None`) is used for interaction events:

```
datatype chanType
  = Int of int
    | Seq of int list
    | String of string
    | Any
    | Tau
    | None ;
```

2.1.2. CSP Operators

Every CSP operator has the deduction rules in the view of the operating semantics in its definition. Following these rules exactly, an ML implementation of a CSP operator can be given. In this subsection, a few examples of the implementation of operators significant to the extension to Timed CSP is discussed.

The *event prefix* $a \rightarrow P$ can be implemented with a higher order function to return an another function from an event and a process:

```
fun prefix (Event (ch, v), P:process) =
  let
    fun temp (Event (ch', v')) =
      if ch' = ch andalso v = v' then P else Bleep
  in
    Proc (temp)
  end
```

In a function definition any variables and functions declared between `let` and `in` are referred locally in the main section between `in` and `end`. In the above case only the function `temp` is defined in this local section. The `Proc` in the main section is declared as a type of function of `event -> process` in the `datatype process` declaration. Thus the main section of the function `prefix` specifies only the `Proc` type function `temp` defined in the local section as a return value. This style to specify only the `Proc` type function as a return is applied commonly in this tool to express CSP operators in ML. The deduction rules for the individual operator is fulfilled in `temp` as above.

The CSP process $a \rightarrow Skip$ is expressed with the above ML notation as:

```
prefix (Event ("a", None), Skip)
```

Since ML can give a function directly as an argument to a function call, a more complex process like $P = b \rightarrow (a \rightarrow SKIP)$ can be expressed using this technique:

```
val P = prefix (Event ("b", None), prefix (Event ("a", None), Skip)) ;
```

For execution of a function returned from CSP operators in ML, we prepare a function like:

```
fun run (Proc temp) = temp
```

This function `run` performs the execution part of the process given in the argument.

The `prefix` can be used for the *output event prefix* with a channel $ch!v \rightarrow P$. For example, a process $ch!10 \rightarrow Skip$ is written as:

```
prefix (Event ("ch", Int 10), Skip)
```

Concerning the *input event prefix* $ch?x \rightarrow P(x)$, it is necessary to keep the input value in the variable x before the event prefix is started. For storage of a value in a channel variable, an associative list is used. An element of the list is a tuple form with (channel_variable, value). The following code indicates a function for the *input event prefix* which is almost identical as `prefix` except the function call of `VarTable.acons`. This function stores a tuple of (variable, value) like `(x,v)` in a list called `varTable` which is actually specified through the input argument. The function `before` means it firstly evaluates `P`, then proceeds to access the database with `VarTable.acons`, and returns the `P` subsequently:

```
fun chanIn (Event (ch, x), varTable, P:Process) =
  let
    fun temp (Event (ch', v)) =
      if ch = ch' then
        P before varTable:=VarTable.acons (x, v, varTable)
      else
        Bleep
  in
      Proc temp
  end
```

The *internal choice* operator $P \sqcap Q$ chooses a process from P and Q via an internal event (τ). The occurrence of internal events can not be controlled from the external environ-

ment. With this operator `Proc P` or `Q` is selected according to the internal event occuring first. The internal events `L` and `R` with their channel type τ (internal event) are introduced. The sequence of the occurrence of the event "L" or "R" will be given randomly at the execution (for the verification) stage. This operator can be described in ML in the following way:

```
fun internal (Proc P, Proc Q) =
  let
    fun temp (Event ("L", Tau)) = Proc P
    | temp (Event ("R", Tau)) = Proc Q
    | temp (x:event) = Bleep
  in
    Proc temp
  end
```

The *external choice* operator $P \,\Box\, Q$ chooses P or Q according to the external event occuring first. If this event is shared with the both processes, the selection becomes non-deterministic, and the internal choice operator is used for this case:

```
fun external (Proc P, Proc Q) =
  let
    fun temp (x:event) =
      if P (x) = Bleep then Q (x)
      else
        if Q (x) = Bleep then P (x)
        else internal (P (x), Q (x))
  in
    Proc temp
  end
```

where `P(x)` and `Q(x)` represent processes P and Q after the event x has occurred.

The *parallel* operator $P \;|[\,A \,\|\, B\,]|\; Q$ means that processes P and Q run concurrently and synchronise with the events that both processes share. The set of events (called alphabets) for P and Q are denoted as A and B respectively in the above example. The events they share is the intersection: $A \cap B$. The implementation is:

```
fun parallel (Proc P, A, B, Proc Q) =
  let
    fun temp (x:event) =
      if isMember (x, A) andalso isMember (x, B) then
         parallel (P (x), A, B, Q (x))
       else if isMember (x, A) then parallel (P (x), A, B, Proc Q)
       else if isMember (x, B) then parallel (Proc P, A, B, Q (x))
       else Bleep
  in
     Proc temp
  end
```

where the function `isMember(x, A)` checks if the event `x` belongs to `A`.

2.1.3. Extension for Timed CSP

In Timed CSP, a process is evolved with time ($P \overset{d}{\rightsquigarrow} P'$), along with CSP conventional labeled transitions such as ($P \xrightarrow{\mu} Q$). For this installation of timing, we have used an idea that the evolution transition is possibly interpreted as one of transitions labeled with time instead of events. With this idea the time can be smoothly included into the existing ML code with the least side effect and modification. This interpretation for the evolution transition forces only slight changes in the declaration of `event` type and in some relevant operators. The declaration of `datatype event` requires a an extra constructor called `Time` as below:

```
datatype event
  = Event of string*chanType
    | Time of Int
```

Timed CSP deals with *real* time values. For simplicity, the type of our new constructor, `Time`, is integer which restricts the timing we can analyse for processes to discrete units only.

The operators in 2.1.2 need to be extended to deal with this modification. For example, the *external choice* is modified to add code for a new deduction rule necessary for Timed CSP. For this operator, the new deduction rule is that the (time) evolution transition of the external choice $P \,\Box\, Q \overset{d}{\rightsquigarrow} P' \,\Box\, Q'$ holds if both $P \overset{d}{\rightsquigarrow} P'$ and $Q \overset{d}{\rightsquigarrow} Q'$ occur:

```
fun external (Proc P, Proc Q) =
  let
    fun temp (Time d) = external (P (Time d), Q(Time d))
    | temp (x:event) =
        if P (x) = Bleep then Q (x)
        else if Q (x) = Bleep then P (x)
        else internal (P (x), Q (x))
  in
    Proc temp
  end
```

The extension of the other operators can are done similarly.

The *timed event prefix* $a@u \rightarrow P$ uses variable u to record the time from the beginning of the process to the event occurrence. A list with element tuples of time variable and actual time `(u,d)` is prepared to associate the time with a variable (in the same way as done in *input event prefix* for channels to hold values received):

```
fun teprefix (Event (ch, v), u, P) =
  let
    fun temp (Time d) =
      teprefix (Event (ch, v), u, P)
      before varTable:=VarTable.acons (u, Time d, varTable)
    | temp (Event (ch', v')) =
      if ch = ch' andalso v = v' then P else Bleep
  in
    Proc temp
  end
```

The *timeout* operator $P \overset{d}{\rhd} Q$ terminates the process P and moves on to Q unless an event of P has occurred within d time. After time d' ($d' < d$), the process goes simply to $P \overset{d-d'}{\rhd} Q$. If time d' exceeds d, then Q starts but it is considered that it has already spent $d' - d$ timing units internally[1]. If the event occurs, then P is made the labeled transition with the event. In a case of the occurrence of an internal event, the event is ignored and the timeout check is kept with time d:

```
fun timeout (Proc P, Time d, Proc Q) =
  let
    fun temp (Time d') =
      if d-d' > 0 then
        timeout (P (Time d'), Time (d-d'), Proc Q)
      else Proc Q (Time d'-d)
    | temp (Event (x, Tau)) =
```

[1]Timed CSP semantics for $\overset{d}{\rhd}$ does not address the meaning of a delay longer than d, but this seems a reasonable extension.

```
        timeout (P (Event (x, Tau)), Time d, Proc Q)
    | temp(e:event) = P (e)
in
  Proc temp
end
```

The *timed prefix* operator $Q = a \xrightarrow{d} P$ means the start of P is forced to wait (paused) for time d after the occurrence of the event a. This process Q can be expressed as $a \rightarrow (STOP \overset{d}{\triangleright} P)$ and we can write this operator using *timeout* as follows:

```
fun tprefix (e:event, Time d, P:process) =
  prefix (e:event, timeout (Stop, Time d, P)
```

3. Refinement with Timed CSP Explorer

The machine-readable CSP (CSP_M) has been developed [9] and is used widely in the applications for CSP. Timed CSP Explorer also uses CSP_M for the machine readable expression of processes written in Timed CSP. We have added four new Timed CSP operators in CSP_M. These notations are given below:

Timed CSP notation	CSP_M
$a@u \rightarrow P$	`a@u -> P`
$P \overset{d}{\triangleright} Q$	`P [>#d Q`
$a \xrightarrow{d} P$	`a ->#d P`
$P \triangle_d Q$	`P /\#d Q`

Then a Timed CSP process $a \xrightarrow{3} (P \overset{5}{\triangleright} Q)$ can be expressed in CSP_M as:

```
a ->#3 ( P [>#5 Q )
```

Timed CSP Explorer has its own lexical analysis and parser parts. The lexical analysis part converts a CSP_M description into a sequence of tokens, and the parser constructs the corresponding ML description (function) from the CSP_M process. The parser part analyses the sequence of tokens into a binary tree, and an ML function for the process is outputted. It is ready to run if arguments are given externally. We have also used the compiler and the programming environment of so-called Standard ML of New Jersey version 110 [7] for the execution of Timed CSP Explorer.

As an example of the refinement of a process with Timed CSP Explorer, Fischer's algorithm (protocol) discussed in detail in [3] is adopted. This algorithm is for the exclusive control of processes to access a shared resource (a critical region). Let's assume the case that the critical region can be accessed by any processes but only one process can access at one time and any other processes should wait till this access is finished. We need an efficient and safe mechanism to control the processes for this case in order not to bring any conflict as each process accesses the critical region independently as if only one process is using it. The Fischer's algorithm is the one of the protocols to describe this exclusive control.

For reason of brevity, we limit the number of processes to two in the demonstration below. A process is signalled to access the shared resource by a request, $req.i$. First, it checks a shared variable V: it can enter the critical region if the value is zero. Otherwise, it must wait until the value becomes zero. If the process finds zero in V, it enters the critical region after writing its own non-zero identification (process) number in V. After exiting the region, it resets V to zero. If we write this algorithm in untimed CSP, it might be expressed in the following way:

$$Q(i) = req.i \rightarrow read?x \rightarrow$$
$$\text{if } x \neq 0 \text{ then } SKIP$$
$$\text{else } write!i \rightarrow enter.i \rightarrow exit.i \rightarrow write!0 \rightarrow SKIP$$

where $1 \leqslant i \leqslant 2$. Initially, we just check the validity of the protocol over a single cycle. We can extend the check to (finitely) many cycles by making $Q(i)$ recursive (where the recursion replaces the $SKIP$s above) and bounding the recursion with a second integer parameter to count down the cycles.

Since two processes $Q(1)$ and $Q(2)$ have no interaction each other, the relation of these processes can be described using the interleave operator as:

$$QS = Q(1) \,|||\, Q(2).$$

The write requests come to the shared variable V, which is a process to maintain a value. V is expressed with the external choice operator: it modifies its value with the received data or outputs the maintained value if a read request is received:

$$V(value) = write?x \rightarrow V(x) \,\Box\, read!value \rightarrow V(value).$$

The main process, FIS, is written:

$$FIS = (QS \;|[\{|\, read, write \,|\}]|\; V(0)) \setminus \{|\, read, write, req \,|\}$$

where V is initialised to zero. The process network for FIS is shown in figure 1.

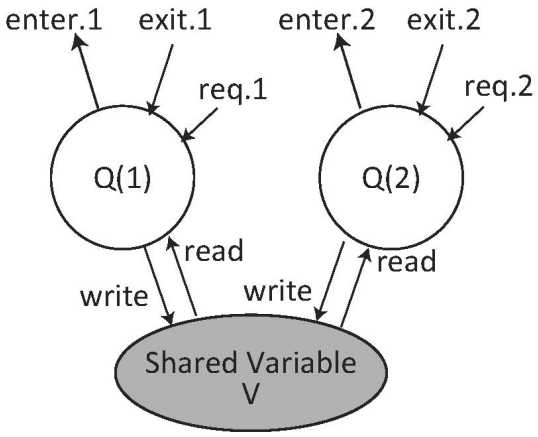

Figure 1. Process network diagram of Fischer's algorithm without time information.

For verification, the specification of the FIS process also needs to be described in CSP. This specification must be such that a consecutive sequence of $enter.i$ events (i=1 or 2) in the trace must not happen:

$$SPEC = (enter.1 \rightarrow exit.1 \rightarrow SPEC \,\Box\, enter.2 \rightarrow exit.2 \rightarrow SPEC).$$

The constraint is made in this formula since either $enter.i$ must be followed by an $exit.i$, so two cannot occur together.

If the trace sets of both processes such that $traces(SPEC) \supseteq traces(FIS)$, then trace refinement is verified – i.e. $SPEC \sqsubseteq_T FIS$ is validated.

Timed CSP Explorer has currently a primitive algorithm for this refinement check. First, all the possible traces are produced by the implementation process and these are stored in a

stack region of the program. Then, the specification process is checked that it allows them step by step. If the specification process terminates without entering into `Bleep` process, the implementation process is a traces refinement of the specification. For timewise refinement, the same procedure is repeated with respected to the same trace set but without timing information. Unless the implementation process has too many state transitions or timing evolutions, this simple algorithm for the verification will work fine.

If a `Bleep` followed by the message of stack overflow occurs, it is difficult to distinguish whether the process being analysed is divergent or simply produces so many events that the stack can not keep all of them before normal termination. Depending on the memory available for Timed CSP Explorer, however, as enough space can be allocated for the stack in the program, most processes terminating with a stack overflow can be suspected as being divergent.

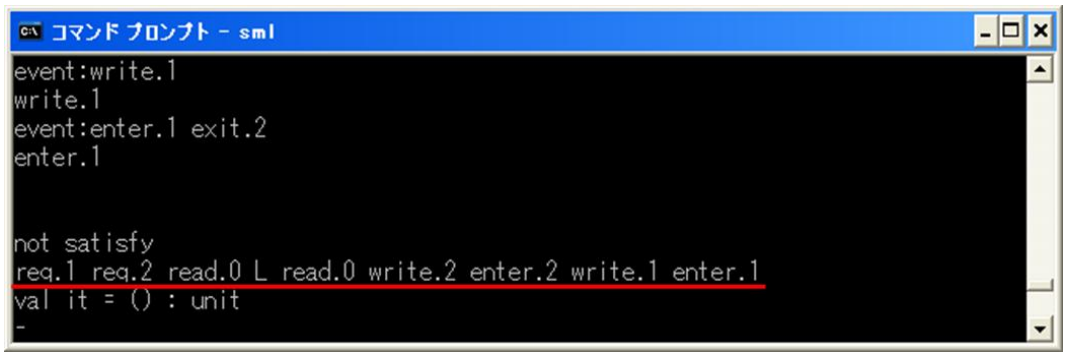

Figure 2. An example output of Timed CSP Explorer for the trace refinement of Fischer's Algorithm described in untimed CSP.

In figure 2, the example of the screen output of the Timed CSP Explorer for this refinement is shown. We find the message "not satisfy" together with a counter example of traces for the refinement. An underline is drawn on the trace to emphasise this. An event with the letter "L" is found in the trace. This "L" means the occurrence of an internal event with an internal choice. This has happened in a trace from the process $Q(1) \;|||\; Q(2)$ after $< req.1, req.2 >$ in this example. This phenomenon can be interpreted that the process FIS has a risk to accept the signal $read.0$ twice at the same time issued independently by $Q(1)$ and $Q(2)$. In this case, an operation of internal choice $Q(1) \sqcap Q(2)$ is performed in the interleaved process. The internal event "L" indicates that the process $Q(1)$ has been chosen in this execution for the verification. If the timing of the check V by either $Q(i)$ is just before the timing to write the non-zero value to V by the other one, both will find 0 in V, and both can enter into the critical region by chance. In this case, consecutive occurrences of $enter.i$ are found with only hidden events (i.e. not an $exit.i$) between them – so the verification is not upheld.

The concept of the time must be introduced for the completion of this algorithm. The Fischer's algorithm is then summarised as,

1. a process $Q(i)$ writes i to V if it finds $V = 0$,
2. it can enter safely into the critical region if it confirms $V = i$ by re-reading V after the time unit ε, and
3. it will never enter into the region unless it finds $V = i$ at this re-reading, this is because the another process must have changed the value during the time span of ε, and the another one is going to work in the critical region.

The new process description of $Q(i)$ with Timed CSP is written as:

$$
\begin{aligned}
Q(i) = \; & req.i \rightarrow read?x \rightarrow \\
& \text{if } x \neq 0 \text{ then } SKIP \\
& \text{else } (write!i \xrightarrow{\varepsilon} read?y \rightarrow \\
& \text{if } y \neq i \text{ then } SKIP \\
& \text{else } enter.i \rightarrow exit.i \rightarrow write!0 \rightarrow SKIP) \overset{\delta}{\triangleright} SKIP.
\end{aligned}
$$

The time ε must be set longer than the time for a process to read, find zero and write a number to V (maximum delay) denoted as δ. If the actual value set in ε is greater than δ, it is expected that the process FIS works fine. This has been confirmed with Timed CSP Explorer for the case of $\varepsilon = 3, \delta = 2$: the output display is not shown because it simply indicates "satisfy". However, if we set $\varepsilon = 2, \delta = 3$, we get the message "not satisfy", and a counter example of traces is shown – see figure 3.

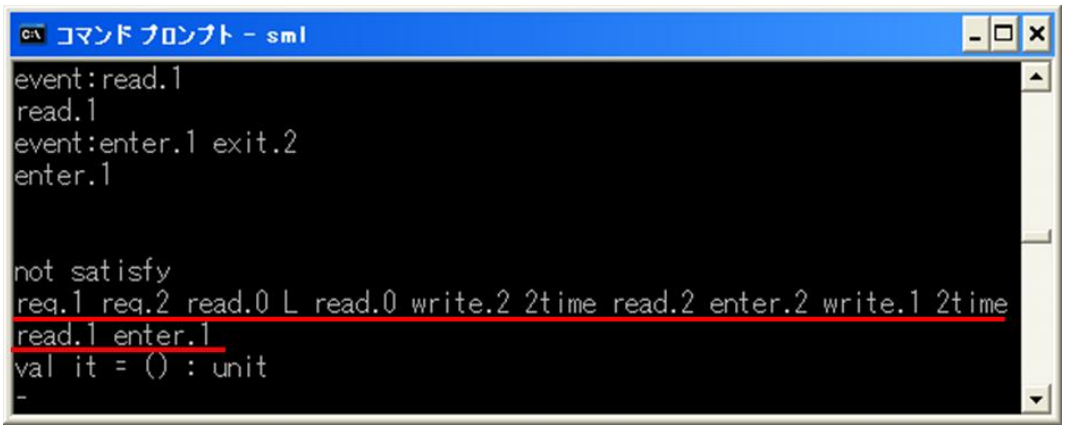

Figure 3. An example output of Timed CSP Explorer for the trace refinement of Fischer's Algorithm described in Timed CSP with an inconsistent timing setting. `2time` shown in the event list represents a time interval of two units.

4. Summary and Outlook

Through this work, we have developed a verification tool called "Timed CSP Explorer" for Timed CSP. We have chosen ML for the programming of CSP processes, and found that the extension to Timed CSP processes could be done quite smoothly with ML. Coding also in ML the lexical analysis and parsing parts for conversion of the (timed) CSP description to ML, we could construct a simple and compact tool for the verification of processes with timed trace and timewise trace refinements.

Since the importance of process description with the concept of the time has become evident (as shown in the example of Fischer's algorithm discussed in section 3), development of a new powerful verification tool for Timed CSP processes (like FDR2 for CSP) has been anticipated. Recently a tool set called PAT (Process Analysis Toolkit) has been developed [5], and prevailed as a general purpose model checker into the community of the formal method. PAT can also make a verification of processes described in Timed CSP. If several tools are developed competitively and used together to check refinements of Timed CSP processes, reliability will be increased. As one such candidate, Timed CSP Explorer will be developed further in future.

Timed CSP Explorer is still limited in the verification of processes, with only trace based refinement currently supported. It has no ability to check failures-based refinements (fail-

ure timewise and timed failures models). Implementation of these refinement procedures is needed for a versatile tool like PAT. It is also required to install an algorithm to compress the number of states and evade the combinatorial explosion of states in process execution for the refinement of large and complicated processes.

Acknowledgements

We would like to express our gratitude to Mr. Kazuto Matsui of NPO CSP Consortium for giving us a hint to develop this tool and various information as well as suggestions and comments during the work. We are also indebted to Prof. Zenjiro Ohba of Toyo University for his generous support and encouragement. We want to thank Dr. Dong Jin Song of the National University of Singapore for giving us the status information of PAT in detail and suggestions for our work.

We thank also all the referees assigned to this document for this conference. They have read carefully and suggested a lot of points to improve this manuscript in terms of not only the scientific contents but also English grammar. Finally we must acknowledge Prof. Peter H. Welch for his enthusiastic support to our work and his apt suggestions.

References

[1] Formal Systems (Europe) Ltd., Failures-Divergent Refinement (FDR2), http://www.fsel.com.
[2] Roscoe, A.W., The Theory and Practice of Concurrency, Prentice Hall International Series in Computer Science. 1997: Prentice Hall.
[3] Schneider, S., Concurrent and Real-Time Systems: The CSP approach. 2000, Wiley.
[4] Orlic B. and J.F. Broenink, “CSP and Real-Time: Reality or Illusion?”, Proceedings of Communication Process Architectures (CPA) 2007 , July 8-11, 2007, Surrey, U.K, p.119.
[5] PAT: Process Analysis Toolkit http://pat.comp.nus.edu.sg.
[6] Paulson, L.C., ML for the Working Programmer (2nd edition), 1996, Cambridge University Press.
[7] Standard ML of New Jersey (SML/NJ), http://www.smlnj.org.
[8] Davis, J. and S.Schneider, “A brief history of Timed CSP”, Theoretical Computer Sciences, **138**, 1995, pp.243-271.
[9] Scattergood, B., “The Semantics and Implementation of Machine-Readable CSP”, Doctor Phil., 1998, Oxford University Computing Laboratory.