A Reasoning Tool for Timed CSP based on Constraint Solving

Jin Song Dong, Ping Hao, Jun Sun, Xian Zhang*
School of Computing,
National University of Singapore
{dongjs,haoping,sunj,zhangxi5}@comp.nus.edu.sg

Abstract

Timed CSP extends CSP by introducing a capability to quantify temporal aspects of sequencing and synchronization. It is a powerful language to model real-time reactive systems. However, there is no verification tool support for proving critical properties over systems modelled using Timed CSP. In this work, we investigate ways of using Constraint Logic Programming (CLP) as an underlying reasoning tool for Timed CSP. We start with encoding the semantics of Timed CSP in CLP, which allows a systematic translation of Timed CSP to CLP. Powerful constraint solver like CLP($\mathcal{R}$) is then used to prove traditional safety properties and beyond, e.g., reachability, deadlock-freeness, timewise refinement relationship, lower or upper bound of a time interval, etc. Finally, we demonstrate the effectiveness of our approach through a case study of the railway crossing system.

1. Introduction

Event-based specification languages like the classic Communicating Sequential Process (CSP) of Hoare's [8] and its timed extension Timed CSP [14], have been proposed for decades. Such specification languages are elegant and intuitive as well as precise. They have been widely accepted and applied to a wide arrange of systems, including communication protocols, embedded systems, etc [15]. Therefore, it is important that system specified using CSP or Timed CSP can be proved formally, and even better if the proving is fully automated.

For CSP, the *de facto* mechanical verification support is its model checker FDR (Failure Divergence Refinement [6, 15]), which verifies safety properties and beyond by showing that there is a refinement relation from the constructed CSP model to the CSP process capturing the properties. There is not yet a mechanized proving method for Timed CSP due to the complexity of time, e.g. the timed trace and failure semantics of Timed CSP is far more complex those of CSP. As far as the authors know, the only attempt is Brooke's work on partial encoding Timed CSP in PVS [2], which relies on heavy user interaction.

In literature, Constraint Logic Programming (CLP [9]) is designed for mechanical proving based on constraint solving. CLP has been applied to model programs and transition systems for the purpose of verification problems [7, 12]. In this work, we propose a constraint-based approach for solving the verification problem of Timed CSP, which readily implies we handle ordinary CSP as well. In contrast to previous works using CLP for specification and verification of real-time systems, our work verifies systems specified using process algebra instead of finite state machines like Timed Safety Automata [1]. The challenge is therefore to cope with the greater expressiveness of Timed CSP and allow efficient automatic proving.

Our approach starts with encoding the semantics of Timed CSP in CLP. Both operational and denotational semantics are encoded, which allows a systematic translation of Timed CSP to CLP. We then go beyond by allowing useful extensions to Timed CSP, for example, the concept of *signal* as in [4] for specifying broadcast communication and some liveness conditions, and integration of Timed CSP and State-based specifications so that we may specify and verify systems with non-trivial data structures.

The practical implication of our translation of Timed CSP to CLP is that powerful constraint solvers like CLP($\mathcal{R}$) [10] can be used to prove properties over systems modelled using Timed CSP. We investigated ways of proving traditional safety properties and beyond, for example reachability, deadlock-freeness, refinement relationship,lower or upper bound of a time interval and etc. The termination of the proving for regular processes are guaranteed by the constraint solving technique called coinductive tabling [11]. We implemented a prototype as a CLP($\mathcal{R}$) program and experimented our encoding with standard real-time systems like the train crossing system.

This work is partially inspired by the work of Jaffar el al. [12]. They presented a CLP proof method for Timed

*Author for correspondence, phone: +65 68742834, fax: +65 67794580

Automata based on a translation of Timed Automata to CLP. They showed that their approach out performs the well-known Timed Automata model checker UPPAAL by allowing a wide range of properties to be verified more efficiently in some cases. In this work, we investigate the idea of using CLP to reason about real-time systems in order to build the first mechanical reasoning tool for Timed CSP. However, because Timed CSP is an event-based process algebra, its expressiveness presents a challenge. For instance, Timed CSP, inherited from CSP, may specify irregular languages, which is certainly beyond the capability of Timed Automata. Moreover, because Timed Automata allow a very restricted form of clock constraints, system behaviors depend on multiple clock valuation at the same time can not be modelled. Timed Automata only handle primitive data components. Our approach however allows proving over systems modelled using irregular processes, with potentially complicated data structures and etc.

The remainder of the paper is organized as follows. Section 2 briefly introduces Timed CSP and the Constraint Logic Programming. Section 3 illustrates the encoding of both operational and denotational semantics of Timed CSP in CLP. A number of useful extensions to Timed CSP are also considered. Section 4 presents various proving we may perform over systems modelled using Timed CSP and translated to CLP. Section 5 illustrates the effectiveness of our approach with a case study. Section 6 concludes the paper.

2. Background

This section is devoted to a brief introduction of Timed CSP and CLP, of which more details can be found in [5] and [9] respectively.

2.1. Timed CSP

Timed CSP extends the well-known CSP (Communicating Sequential Processes) notation of Hoare [8] with timing primitives. CSP is an event based notation primarily aimed at describing the sequencing of behavior within a process and the synchronization of behavior (or *communication*) between processes. Timed CSP extends CSP by introducing a capability to quantify temporal aspects of sequencing and synchronization. Inherited from CSP, Timed CSP adopts a symmetric view of process and environment. Events represent a cooperative synchronization between process and environment. Both process and environment may control the behavior of the other by *enabling* or *refusing* certain events and sequences of events.

The syntactic class of Timed CSP expressions is defined as the following:

$$
\begin{aligned}
P ::= \ & \textsc{Stop} \mid \textsc{Skip} \mid \textsc{Run} \\
& \mid e \xrightarrow{t} P \mid e : E \to P(e) \mid e \bullet t \to P(t) \\
& \mid P_1 \,\Box\, P_2 \mid P_1 \sqcap P_2 \\
& \mid P_1 \ {}_X||_Y \ P_2 \mid P_1 \,|[X]|\, P_2 \mid P_1 \,|||\, P_2 \\
& \mid P_1;\ P_2 \mid P_1 \,\triangledown\, P_2 \mid P1 \,\triangleright\{d\}\, P2 \\
& \mid \textsc{Wait}[d] \mid P_1 \,\triangledown\{d\}\, P_2 \mid \mu X \bullet P(X)
\end{aligned}
$$

$\textsc{Run}_\Sigma$ is a process always willing to engage any event in Σ. STOP denotes a process that deadlocks and does nothing. A process that terminates is written as SKIP. A process which may participate in event e then act according to process description P is written as $e \bullet t \to P(t)$. The event e is initially enabled by the process and occurs as soon as it is requested by its environment, all other events are refused initially. The (optional) timing parameter t records the time, relative to the start of the process, at which the event e occurs and allows the subsequent behavior P to depend on its value. The process $e \xrightarrow{t} P$ delays process P by t time units after engaging event e.

Diversity of behavior is introduced through two choice operators. The external choice operator ($\Box$) allows a process of choice of behavior according to what events are requested by its environment. For instance, the process $(a \to P) \,\Box\, (b \to Q)$ begins with both a and b enabled. The environment chooses which event actually occurs by requesting one or the other first. Subsequent behavior is determined by the event which actually occurred. Internal choice represents variation in behavior determined by the internal state of the process. The process $a \to P \sqcap b \to Q$ may initially enable either a or b or both, as it wishes, but must act subsequently according to which event actually occurred. The environment cannot affect internal choice.

The parallel composition of processes P_1 and P_2, synchronized on common events of their alphabets X, Y (or a common set of events A) is written as $P_1 \ {}_X||_Y \ P_2$ (or $P_1 \,|[A]|\, P_2$). No sharing event may occur unless enabled jointly by both P_1 and P_2. When a sharing event does occur, it occur in both P_1 and P_2 simultaneously and is referred to as *synchronization*. Events not sharing may occur in either P_1 or P_2 separately but not jointly.

The sequential composition of P_1 and P_2, written as $P_1;\ P_2$, acts as P_1 until P_1 terminates by communicating a distinguished event $\checkmark$ and then proceeds to act as P_2. The termination signal is hidden from the process environment and therefore occurs as soon as enabled by P_1. The interrupt process $P_1 \,\triangledown\, P_2$ behaves as P_1 until the first occurrence of event in P_2, then the control passes to P_2. The timed interrupt process $P_1 \,\triangledown\{d\}\, P_2$ behaves similarly except P_1 is interrupted as soon as d time units have elapsed.

A process which allows no communications for period d time units then terminates is written as $\textsc{Wait}[d]$. The timeout construct written as $P_1 \,\triangleright\{d\}\, P_2$ passes control to an exception handler P_2 if no event has occurred in the primary process P_1 by some deadline d. Recursion is used to

give finite representation of non-terminating processes. The process expression $\mu X \bullet P(X)$ describes a process which repeatedly act as $P(X)$.

In general, the behavior of a process at any point in time may be dependent on its internal state and this may conceivably take an infinite range of values. It is often not possible to provide a finite representation of a process without introducing some notation for representing this internal state. The approach adopted by Timed CSP is to allow a process definition to be parameterized by state variables. Thus a definition of the form $P(x)$ represents a family of definitions, one for each possible value of *x*.

The semantics of a Timed CSP process is precisely defined either by identifying how the process may evolve through time or by engaging in events (operational semantics [17]) or by stating the set of observations, e.g., traces, failures and timed failures (denotational semantics [4]).

Example We take a simple timed vending machine as an example. A user may insert some coins and then make a choice between coffee or tea. Once the choice is made, the vending machine dispatches the corresponding drink. Or the user may ask the machine to release the coins and walk away. If the user idles more than 10 seconds after the coin is inserted, the machine will release the coins.

$$
\begin{aligned}
TVM \mathrel{\widehat{=}} \mu X \bullet\ & insert \rightarrow \\
& ((reqrelease \rightarrow release \xrightarrow{2} X) \\
& \Box\ (coffee \xrightarrow{3} dispatchcoffee \rightarrow X) \\
& \Box\ (tea \xrightarrow{2} dispatchtea \rightarrow X)) \\
& \triangleright\{10\}\ (release \rightarrow X)
\end{aligned}
$$

2.2. CLP Preliminaries

Constraint Logic Programming (CLP) began as a natural merger of two declarative paradigms: constraint solving and logic programming. This combination helps make CLP programs both expressive and flexible. The CLP scheme defines a class of languages based upon the paradigm of rule-based constraint programming. Each instance of the scheme is a programming language and is obtained by specifying a structure of computation. That is, the domain of discourse and the functions and relations on this domain characterize the language.

A *constraint* is written using a language of functions and relations, which are used in the basic programming language to describe expressions and conditions. in CLP($\mathcal{R}$), it can be either an *arithmetic constraint* or *primitive constraint*. A primitive constraint is *solvable* iff there is an appropriate assignment of real numbers and ground terms to the variables such that the constraint evaluates to true.

The *universe of discourse* $\mathcal{D}$ of our CLP program is a set of terms, which can be either simple terms or compound terms constructed from simple terms. The simple terms are variable terms, numeric constant terms, symbolic numeric constants, functor constant terms and string constant terms.

A compound term can be a list of the notation: [L]. The other compound term is an *atom*, of the form $p(\tilde{t})$, where *p* is a user defined predicate symbol and $\tilde{t}$ is a sequence of terms. The set of $p(\tilde{d})$ where p ranges over the predicates and $\tilde{d}$ ranges over the tuples in $\mathcal{D}$ is called the *domain base* $\mathcal{B}$. A *rule* is of the form $A_0 : -\alpha_1, \alpha_2, ..., \alpha_k$, where each α_i is either a primitive constraint or an atom. The atom A_0 is the *head* of the rule while the remaining primitive constraints and atoms constitute the body of the rule. A *goal* has exactly the same format as the body of the rule of the form $? - \alpha_1, \alpha_2, ..., \alpha_k$. In case there are no atoms in the body, we may call the rule a *fact*. All goals, rules and facts are terms.

Inherent in the operational model of CLP is a *subgoal selection strategy*, which selects a constraint or an atom from a given goal. Now let P denotes a CLP program and G be a goal with the form: $A_1, A_2, ..., A_n$, n$\geq$ 0, whose subsequence of solved constraints are denoted by $\sigma_1, \sigma_2, ..., \sigma_m$, m$\geq$ 0, and whose subsequence of delayed constraints are denoted by $\delta_1, \delta_2, ...\delta_k$, k$\geq$ 0. We say that there is a *derivation step* from G to another goal G_2 if one of the following holds:

- The subgoal selection strategy selects δ_i, $1 \leq i \leq k$, from G. In G_2, the subsequence of atoms are $A_1, A_2, ..., A_n$, the subsequence of solved constraints are $\sigma_1, \sigma_2, ..., \sigma_m, \delta_i$, the subsequence of delayed constraints are $\delta_1, ..., \delta_{i-1}, \delta_{i+1}, ..., \delta_k$. Furthermore, the conjunction of the solved constraints in G_2 is solvable.

- The subgoal selection strategy selects A_i,$1 \leq i \leq k$, from G, and the program P contains rule R which can be renamed so that it contains only new variables and takes the form: $B : -B_1, B_2, ..., B_s, \delta'_1, \delta'_2, ...\delta'_t$. $s \geq 0, t \geq 0$, where B_i denotes the atoms and δ'_i denotes the constraints. In G_2, the subsequence of the atoms are $A_1, ..., A_{i-1}, B_1, ..., B_s, A_{i+1}, ..., A_n$, the subsequence of the solved constraints are $\sigma_1, \sigma_2, ..., \sigma_m$, the subsequence of the delayed constraints are $\delta_1, ...\delta_k, A = B, \delta'_1, ...\delta'_t$. Furthermore, the conjunction of the solved constrains in G_2 is solvable.

We say that the delayed constraints δ_i or atom A_i in G is the *selected subgoal*. Equivalently, δ_i or A_i is the subgoal of G chosen to be *reduced*.

A *derivation sequence* is a possibly infinite sequence of goals, starting with an initial goal. A sequence is successful if it is finite and the last goal G_n with no atoms which is called a *terminalgoal*, and the constraints in G_n are called *answer constraints*.

3. Timed CSP Semantics in CLP

This section is devoted to an encoding of the semantics of Timed CSP in CLP. The practical implication is that we may then use powerful constraint solver like CLP(R) [10] to do various proving over systems modelled using Timed CSP. Both the operational semantics and denotational semantics are encoded, among which the encoding of operational semantics serves most of our purposes, nevertheless the encoding of the denotational semantics offers an alternative way of proving systems modelled in Timed CSP as well as the correctness of the encoding itself. For example, we may prove the event history of the operational semantics model is always allowed in the denotational semantics model.

The very initial step of our work is the syntax encoding of Timed CSP process in CLP syntax, which can be automated easily by syntax rewriting. For instance, Figure 1 is the syntax encoding of process *TVM* in CLP.

3.1. Operational Semantics

The operational semantics of Timed CSP is precisely defined by Schneider [17] using two relations: an evolution relation and a timed event transition relation. It is straightforward to verify that our encoding conforms the two relations in [17].

A relation of the form *tos(P1,T1,E,P2,T2)* is used to capture the operational semantics. Informally speaking, *tos(P1,T1,E,P2,T2)* is true if the process *P1* may evolve to *P2* through either a timed transition, i.e., let *T2-T1* time units pass, or an event transition by engaging an abstract event instantly[1]. The relation *tos* defines a transition system interpretation of a Timed CSP process, where the state is identified by the combination of the the process expression and the time variable. Using tabling mechanism offered in some of the constraint solvers like CLP($\mathcal{R}$) [10] or XSB [19], the termination of the derivation sequence based on relation *tos* depends on the finiteness of the reachable process expressions from the initial one[2]. Therefore, if a process is irregular, proving of goals which need to explore all reachable process expressions is not feasible. However, even for irregular processes, interesting proving like existence of a trace is still possible.

We define the *tos* relation in terms of each and every operator of Timed CSP. For the moment, we assume the process is not parameterized and we shall handle parameterized processes uniformly in Section 3.3. For instance, the primitives process expressions in Timed CSP is defined through the following clauses.

[1]Or both at the same time by engaging an nontrivial action which takes time (necessary for only extensions to Timed CSP like TCOZ [13] where E could be a complicated computation)

[2]Trivial time steps can be aggressively abstracted, for example, a trivial time step is allowed only if the enabled events are changed.

```
tos(stop,T1,[],stop,T2) :- D>=0, T2=T1+D.
tos(skip,T,[termination],stop,T).
tos(skip,T1,[],skip,T1+D) :- D>=0.
tos(run,T,[_],run,T).
tos(run,T1,[],run,T2) :- D>=0, T2=T1+D.
```

The only transition for process STOP is time elapsing. Whereas process SKIP may choose to wait some time before engaging event *termination* which is our choice of representation for event $\checkmark$ in CLP. Process RUN may either let time pass or engaging any event. In the following, we show how hierarchical operators are encoded in CLP using the alphabetized parallel composition operator as an example.

In the operational semantics, the event transition and evolution transition associated with the alphabetized parallel composition operator is illustrated as the following [17]:

$$\frac{P_1 \xrightarrow{e} P_1'}{P_1 \;{}_X||_Y\; P_2 \xrightarrow{e} P_1' \;{}_X||_Y\; P_2} [\, e \in X \cap \{\tau\} \setminus Y \,]$$

$$\frac{P_2 \xrightarrow{e} P_2'}{P_1 \;{}_X||_Y\; P_2 \xrightarrow{e} P_1 \;{}_X||_Y\; P_2'} [\, e \in Y \cap \{\tau\} \setminus X \,]$$

$$\frac{P_1 \xrightarrow{e} P_1', P_2 \xrightarrow{e} P_2'}{P_1 \;{}_X||_Y\; P_2 \xrightarrow{e} P_1' \;{}_X||_Y\; P_2'} [\, e \in X \cap Y \,]$$

$$\frac{P_1 \overset{d}{\rightsquigarrow} P_1', P_2 \overset{d}{\rightsquigarrow} P_2'}{P_1 \;{}_X||_Y\; P_2 \overset{d}{\rightsquigarrow} P_1' \;{}_X||_Y\; P_2'}$$

The $\rightarrow$ represents an event transition, whereas $\rightsquigarrow$ represents an evolution transition. The rules associated with the alphabetized parallel composition operator is as the following:

```
tos(para(P1,P2,X,Y),T,[E],para(P3,P2,X,Y),T)
    :- tos(P1,T,[E],P3,T),
       member(E,X), not(member(E,Y)).
tos(para(P1,P2,X,Y),T,[E],para(P1,P4,X,Y),T)
    :- os(P2,T,[E],P4,T),
       member(E,Y), not(member(E,X)).
tos(para(P1,P2,X,Y),T,[E],para(P3,P4,X,Y),T)
    :- tos(P1,T,E,P3,T),tos(P2,T,E,P4,T),
       member(E,X), member(E,Y).
tos(para(P1,P2,X,Y),T1,[],para(P3,P4,X,Y),T1+D)
    :- tos(P1,T1,[],P3,T1+D),
       tos(P2,T1,[],P4,T1+D).
```

The first two rules state that either of the components may engage an event as long as the event is not shared. The third rule states that a shared event can only be engaged simultaneously by both components. The last expresses that the composition may allow time elapsing as long as both the components do. Other parallel composition operation, like $|[X]|$ and $|||$, can be defined as special cases of the alphabetized parallel composition operator straightforwardly.

There is a clear one-to-one correspondence between these rules and the event transitions and evolution transitions illustrated above. So are the rules associated with

```
proc(c1,delay(coffee, eventprefix(dispatchcoffee, tvm), 3)).
proc(c2,delay(tea, eventprefix(dispatchtea, tvm), 2)).
proc(c3,eventprefixc(reqrelease, delay(release,tvm,2))).
proc(choices,extchoice(extchoice(C,T),R)):- proc(c1,C), proc(c2, T), proc(c3 ,R).
proc(to, timeout(C,eventprefix(release, tvm),10)):- proc(choices,C).
proc(vending, recursion([tvm, eventprefix(insert, P)], eventprefix(insert, P))):- proc(to, P).
```

Figure 1. Vending Machine in CLP

other compositional operators, which are partly illustrated in Figure 2 and fully at our website[3]. Therefore, the soundness of the encoding can be proved by showing there is a bi-simulation relationship between the transition system interpretation defined in [17] and ours. With the clear correspondence between our rules and the transitions defined in [17], the bi-simulation relationship can be proved easily via a structural induction.

For simplicity, we do restrict the form of recursion to $\mu X \bullet P(X)$, which means mutual recursion through process referencing has to been transformed before hand. The following clauses illustrate how recursion is handled, where N is the recursion point, i.e., X in $\mu X \bullet P(X)$) and P is the process expression, i.e., $P(X)$.

```
tos(recursion([N,P],P1),T,[E],
        recursion([N,P],P2),T)
    :- not(P1==N), tos(P1,T,[E],P2,T).
tos(recursion([N,P],N),T,[],
        recursion([N,P],P),T).
tos(recursion([N,P],P1),T1,[],
        recursion([N,P],P2),T1+D)
    :- D>0, tos(P1,T1,[],P2,T1+D).
```

3.2. Denotational Semantics

We also encode both the timed traces and the timed failures model of Timed CSP, where the semantics of a Timed CSP process is represented by a set of timed traces or a set of timed failures [16]. In contrast to the operational semantics, which focuses on a single step at once, the denotational semantics captures all possible observations of systems modelled using Timed CSP. Therefore, it is easier to prove over all possible behaviors in the denotational semantics model. In the following, we illustrate our encoding using only a few fundamental constructors for the sake of space saving.

A relation *timedfailure(P, f(Tr, R))* is defined to capture the timed failure semantics, where *P* is a process expression and *Tr* is a sequence of timed events and *R* is a set of timed refusals. For instance,

```
timedfailure(stop, f([],_)).
timedfailure(skip, f([], R))
    :- sigma(R,S), not(member(termination,S)).
timedfailure(skip,f([tevent(T,termination)],R))
    :- T>=0, before(R,T,Z), sigma(Z,N),
       not(member(termination, N)).
```

[3]http://nt-appn.comp.nus.edu.sg/fm/clp

The relation *sigma(P, S)* is used to retrieve all events S in a process expression P, i.e., $S = \sigma(P)$. Similarly, the relation *before(R, T, Z)* is defined accordingly as $Z = R \upharpoonright T$, i.e., the refusals before time T. Basically, the first rule states that the failures of process STOP are an empty trace with all possible refusals. Whereas the process SKIP refuses everything until the occurrence of event *termination*, and all events are refused afterwards. As for compositional operators, we take the interface parallel composition operator as an example.

```
timedfailure(parallel(Q1,Q2,A), failure(S,N))
  :- timedfailure(Q1, failure(S1,N1)),
     timedfailure(Q2, failure(S2,N2)),
     union(N1,N2,N),
     union(A,[termination],AT),
     remove(N1,AT,N11), remove(N2,AT,N22),
     setequal(N11,N22), tsynch(S1,S2,A,S).
```

The relation *union(X, Y, Z)* is the set union, i.e., $Z = X \cup Y$. The relation *remove(X, Y, Z)* is the set subtraction, i.e., $Z = X \setminus Y$. The relation *tsynch* defines the ways in which a trace tr_1 from component $Q1$ and a trace tr_2 from component $Q2$ can be combined to form a trace of the parallel (refer to [16] for the formal definition). The interface parallel operator combines the features of the interface parallel and the interleaving operators, requiring synchronization on events from the interface event set A, and interleaving on events not in A. This means that events within the set A can be refused by either processes, whereas events outside A can only be refused when both processes refuse them.

Notice that while the operational semantics focuses on how the system is evolved through either time or event's occurrence, the denotational semantics focuses on observations of the system, which allows us to query the system behaviors as a whole. For instance, it is more straightforward to check timewise refinement using the denotational semantics, and irregular processes can be handled if we replace the recursion using its fixed point. However, because there is no guarantee that the derivation sequence is terminating, we have to limit the height of the proving tree. For example, one possible way that has been implemented is to limit the number of times a recursion is resolved.

3.3. Handling Extensions to Timed CSP

Timed CSP is introduced as a rather useful extension to hoare's CSP in [14]. Since then, various extensions of

```
tos(eventprefix(E,P),T,[E],P,T).
tos(eventprefix(E,P),T1,[],eventprefix(E,P),T1+D) :- D>=0.
tos(prefixchoice(X,P),T,[Y],P,T) :- member(Y,X).
tos(prefixchoice(_,P),T1,[],P,T1+D) :- D>=0.
tos(timeout(Q1,_,_),T,[E],P,T) :- tos(Q1,T,[E],T,P).
tos(timeout(Q1,Q2,D),T1,[],timeout(P,Q2,D-T),T1+T) :-T>=0,T<=D,tos(Q1,T1,[],P,T1+T).
tos(timeout(Q1,Q2,D),T,[tau],timeout(P,Q2,D),T) :-tos(Q1,T,[tau],P,T). tos(timeout(_,Q2,0),T,[tau],Q2,T).
tos(wait(D),T1,[],P,T2) :- tos(timeout(stop,skip,D),T1,[],P,T2).
tos(extchoice(P1,_),T,[E],P3,T) :- tos(P1, T, E, P3, T).
tos(extchoice(_,P2),T,[E],P4,T) :- tos(P2,T,E,P4,T).
tos(extchoice(P1,P2),T1,[],extchoice(P3,P4),T1+D):-D>=0,tos(P1,T1,[],P3,T1+D),tos(P2,T1,[],P4,T1+D).
tos(hide(P1,X),T,[tau],hide(P2,X),T) :- tos(P1,T,[E],T,P2),member(E,X).
tos(hide(P1,X),T,[E],hide(P2,X),T) :-tos(P1,T,[E],P2,T), not(member(E,X)).
tos(hide(P,X),T1,[],hide(Q,X),T1+D):-D>=0,tos(P,T1,[],Q,T1+D),not(member(A,X),tos(P,_,[A],_,_)).
tos(sequ(P1,P2),T,[E],sequential(P3,P2),T) :- tos(P1,T,[E],P3,T),not(E=termination).
tos(sequ(P1,P2),T,[termination],P2,T) :-tos(P1,T,[termination],_,T).
tos(sequ(P1,P2),T1,[],sequ(P3,P2),T2):-T2>=T1,tos(P1,T1,[],P3,T2),not(tos(P1,_,[termination],_,_)).
tos(interrupt(P1,P2),T,[E],interrupt(P3,P2),T) :-tos(P1,T,[E],P3,T).
tos(interrupt(_,P2),T,[E],P3,T) :-tos(P2,T,[E],P3,T).
tos(interrupt(P1,P2),T1,[],interrupt(P3,P4),T1+D):-D>=0,tos(P1,T1,[],P3,T1+D),tos(P2,T1,[],P4,T1+D).
tos(tinterrupt(Q1,Q2,D),T,[E],tinterrupt(Q3,Q2,D),T) :-tos(Q1,T,[E],Q3,T), not(E==termination).
tos(tinterrupt(Q1,_,_),T,[termination],Q3,T) :-tos(Q1,T,[termination],Q3,T).
tos(tinterrupt(_,Q2,0),T,[tau],Q2,T).
tos(tinterrupt(Q1,Q2,D),T1,[],tinterrupt(Q3,Q2,D),T1+T) :-T>=0,T<=D, tos(Q1,T1,[],Q3,T1+T).
```

Figure 2. Operational Semantics of Timed CSP in CLP

Timed CSP has been proposed. In this work, we identify some of the extensions that we believe is effective in modelling systems and show that they can be encoded in the CLP framework. For instance, the idea of *signal* by Davies [4] is a simple yet useful extension to capture liveness as well as model broadcasting effectively. The motivation of the concept *signal* is that when describing the behavior of a real-time process, we may wish to include instantaneous observable events that are not synchronization. For example, an audible bell might form part of the user interface to a telephone network, even though the bell may ring (a *signal*) without the cooperation of the user. Informally, *signal* events are distinguished events that will occur as soon as they become available, and will propagate through parallel composition. A process may ignore any signal performed by another process, unless it is waiting to perform the corresponding synchronization. The semantics for timed signals model requires that for any observation can be extended into the future, the only events must be observed are signals. Therefore, signals are useful both for modelling broadcast communication and specifying liveness conditions, i.e., some events must be engaged. We encode the denotational semantic model for timed signals, in which relation $sigTF(P, sigfailure(Tt, Tr, T))$ is used to capture this time failure semantics for signals, where P denotes the process, Tt is the timed trace, Tr denotes the timed refusal set and T denotes a time value. The following CLP clauses illustrate the possible evolution of signal event prefixing.

```
sigTF(eventprefix(E,_,_),sigfailure([],X,T))
  :- not(E==sig(_)), sigma(X,Z),
     not(member(E,Z)),end(X,T1), T>=T1.
sigTF(eventprefix(E,P,D),
      sigfailure([tevent(T,E)|XS],Y,T1+D+T))
   :- T>=0, not(E==sig(_)),
      sigTF(P,sigfailure(S,Y1),T1),
      backthrough(Y,T+D,Y1), begin(S,T2), T2>=T+D,
      end(S,Y,T3), max(T,T3,T4), T1+D+T >= T4,
      before(Y, T, Z), sigma(Z,N), not(member(E,N)),
      delay(S,T+D,XS).
sigTF(eventprefix(sig(E), P, D), sigfailure([],[],0)).
sigTF(eventprefix(sig(E),P,D),
      sigfailure([tevent(0,E)|XS],Y,T))
  :- sigTF(P,failure(S,Y1),T1), backthrough(Y,T+D,Y1),
   T=T1+D, before(Y,T,Z), sigma(Z,N),
   not(member(E,N)), delay(S,T+D,XS).
```

The first two clauses denote the semantics for event prefix process $a \rightarrow P$ where a is not a signal, while the last two denote the one with signal event $\hat{a}$, presented as $sig(a)$. In the above rules, *end(X,T)* computes the least upper bound of the time refusal X. *backthrough(Y,T,Y1)* represents the relation: Y - T = Y1, i.e., timed refusal $Y1$ is generated from Y by translating it backwards through time T. *begin(S, T)* retrieves the time of the first event in timed trace S.

Another extension of special interest is Timed CSP integrated with state-based languages like Z [20] to model systems with not only complicated control flow but also complex data structures [13, 18]. Instead of adopting a heavy language like TCOZ (Timed Communicating Object-Z [13]), we allow a finite number of variables to be associated with a process[4], called state variables. In addition, we allow a state update transition, i.e., instead of engaging an abstract event, the system may perform a state update which changes the valuation of the state variables. A state update is specified as a predicate involving state variables before and after the update, as in Z style where the after-variables are primed [20].

For instance, there is a fragment of the specification of

[4] which are of types supported by current tools for CLP.

this vending machine, in which we allow different coins to be inserted via a channel communication *insert?x* where *x* is 10, 20 or 50, a data variable *Quota* is requested to accumulate the amount of all coins inserted by the user.

$$Insert(Quota) \mathrel{\widehat{=}} insert?x \rightarrow AddQuota$$

This Timed CSP specification corresponds to the following CLP clauses where both the pre and post values of the process parameter are presented as the parameters, namely Quota1 and Quota2 in this example, of the relation *proc*. The user is responsible to specify exactly how an action updates the data variables, e.g., adding the amount of the coin to *Quota*.

```
proc(insert,eventpreifx(insert(X1),aq),Quota1,Quota2)
    :- action(aq, X1, Quota1, Quota2).
action(aq,X1,Quota1,Quota2) :- Quota2 = Quota1 + X1.
```

4. Proving Timed CSP

This section is devoted to show the various proving we may perform over systems modelled using Timed CSP and then encoded in CLP. We implemented a prototype in one of the CLP solver, namely CLP(R). Any CLP assertion can be proved against a given real-time system. We also developed a number of shortcuts for easy querying and proving.

4.1. Safety and Liveness

Using CLP, we may make explicit assertion which is neither just a safety assertion, nor just a liveness assertion. Yet it can be used for both purposes using a unique interpretation. In the following, we show how safety properties and liveness properties, like reachability, can be queried. We employ the concept of *coinductive tabling* with the purpose of obtain termination when dealing with recursions, which facilitates verifying safety and liveness properties based on traces. The detailed introduction of *coinductive tabling* can be found in [11].

Because Timed CSP is an event-based specification language, it is clearly useful to prove safety and liveness properties in terms of predicate concerning not only state variables but also events. A discussion on how to allow such temporal properties is presented in [3]. In order to explore the full state space, we define the following[5]:

```
treachable(P, P, T1, T1).
treachable(P, Q, T1, T2)
   :- tos(P,T1,_,P1,T3),treachable(P1,Q,T3,T2).
```

The relation *treachable(P, Q, T1, T2)* states that it is possible to reach the process expression *Q* at time *T2* from *P* at time T1. By using the tabling method, we dynamically record the process expressions that have been explored so as to avoid re-exploring them. In this regard, one kind of liveness property namely reachability is easily asserted using *treachable*.

An invariant property (a predicate over time variable and state variables and possible local clocks) is in general expressed as the assertion:

```
inv(P, T, Property)
    :- not(treachable(P, Q, T, T1),
       not(sat(Property))).
```

One safety property of special interest is deadlock-freeness. The following clauses are used to prove it.

```
tdeadlock(P,T1) :-
  treachable(P,P1,T1,T2),
  (not(tos(P1,T2,[],Q,T), tos(Q,T,[_],_,_));
  (tos(P1,T2,[],Q,_); not(tos,P1,T2,[_],_,_))),
  printf("deadlock at: %\n",[P1]).
```

Basically, it states that a process *P* at time *T1* may result in deadlock if it can reach the process expression *Q* at time *T2* where no event transition is available neither at *T2* nor at any later moment. The last line outputs the deadlocked process expression. Alternatively, we may present it as a result of the deadlock proving.

We allow trace-based properties (safety or liveness[6]) that can be checked by exploring trace set partially. The retrieve of a trace is done by the predicate $superstep(P, N, Q)$, which finds a sequence of events through which process expression *P* evolves to *Q*:

```
superstep(P,[-],_)
    :- not(tos(P,_,_,Q,_), not table(Q)).
superstep(P,[A|N],Q)
    :- tos(P,_,M,P1,_), not(M==[];M==[tau]),
       M=[A], not table(P1),
       assert(table(P1)), superstep(P1,N,Q).
superstep(P,N,Q)
    :- tos(P,_,M,P1,_), (M==[];M==[tau]),
       not table(P1), assert(table(P1)),
       superstep(P1,N,Q).
```

We may prove that some event will always eventually be ready to be engaged using the following rule: where rule $member(N, E)$ returns true if event *E* appears at least once in the event sequence *N*.

```
finally(P,E) :- not(superstep(P,N,_)),not member(N,E)).
```

Predicate $finally(P, E)$ captures the idea that there is no such trace without event *E* in this process *P*. In other words, this process will eventually go to event *E*. Another property based on traces would be identifying the relationship among events, e.g. event *A* can never happen before (after) event *B* in a trace or trace fragment. Take the timed vending for example, we would like to ensure that in a round of using the machine, the event *tea* will never be followed by an event *dispatchcoffee*.

[5] The possible state variables and local clocks are skipped for simplicity.

[6] Liveness in CSP is a bit different from that in traditional temporal logic. In CSP, a process may wait forever before engaging an available event.

Example For the timed vending machine, we would like to check that it is deadlock-free by running the following goal and expecting failure:

$$?- proc(vending, P), tdeadlock(P, 0)$$

Moreover, we would expect that whenever we choose *tea*, it would never dispatch *coffee* instead of *tea*, which can be checked by the following goal:

$$?- proc(vending, P), super(P, N), (not\ in(tea, N); after(n, dispatchcoffee, tea)).$$

4.2. Timewise Refinement Checking

The notion of refinement is a particularly useful concept in many forms of engineering activity. If we can establish a relation between components of a system which captures the fact that one satisfies at least the same conditions as another, then we may replace a worse component by a better one without degrading the properties of the system. Compared to untimed CSP refinements which can be defined by FDR [15], timedwise refinements for Timed CSP contain more infromation about timing behavior. With the denotational model - timed failure model build in CLP, the refinement relations can be defined for systems described in Timed CSP in several ways, depending on the semantic model of the language which is used. In the timed versions of CSP, we mainly concentrate on two forms of refinement, corresponding to the semantic models which are trace timewise refinement and failure timewise refinement.

Trace timewise refinement Classical refinement in the traces model states that P_1 is refined by P_2 if any trace in P_2 is also a trace in P_1. Following this approach, a process Q is a trace timewise refinement of P if all of its timed traces are allowed by P. The trace timewise relation is written $P\ {}_T\sqsubseteq_{TF}\ Q$ where P is an untimed CSP process, and Q is a timed CSP process. It is defined as:

$$P\ {}_T\sqsubseteq_{TF}\ Q = \forall(s, \aleph) \in \mathcal{TF}[\![\mathcal{Q}]\!] \bullet \#s < \infty \Rightarrow strip(s) \in traces(P)$$

In our timed failure model in CLP, we are able to find any finite timed trace of a process. Instead of testing every timed trace of a process Q by proving that this timed trace s with times removed is also a legal trace for the untimed process P, we test on the negation of this predicate. We introduce the predicate *traceTR* to find a violative timed trace of Q that is not a legal trace of P with its time information removed. The definition of *timedTR* is given by the following CLP clause: where Q is the timed process, P is the untimed process, S is a timed trace of Q and *TimeRmTr* represents the times removed version of S.

```
traceTR(P,Q,S)
    :- timedfailure(Q, failure(S,Refusal)),
       strip(S,TimeRmTr),
       not(trace(P,TimeRmTr)).
```

Failures timewise refinement The timed process Q will be a failure timewise refinement of the untimed process P if all of its timed traces are allowed by P, as well as all its timed failures area allowed by the stable failures of P. This is formally defined as:

$$P\ {}_{SF}\sqsubseteq_{TF}\ Q = \forall(s, \aleph) \in \mathcal{TF}[\![\mathcal{Q}]\!] \bullet \#s < \infty \Rightarrow strip(s) \in trace(P) \wedge (\exists\, t : R^+;\ X \subseteq \Sigma \bullet ([t, \infty) \times X) \subseteq \aleph \Rightarrow (strip(s), X) \in \mathcal{SF}[\![\mathcal{P}]\!])$$

We take the similar approach as the trace timewise refinement which tests the negation of the universal predicate. The predicate *failureTR* is introduced to capture this idea, that can be represented by the following CLP clauses:

```
failureTR(P,Q,S,Refusal)
   :- timedfailure(Q,failure(S,Refusal)),
      ((strip(S,TimeRmTr),
      not(trace(P, TimeRmTr)));
      not(inStableFailure(Q, S,Refusal,P))).
inStableFailure(Q,S,Refusal,P)
   :- T>0, sigma(Q, Sigma), subset(Sigma,X),
      (not(subset(prod(int(T,inf),X),Refusal));
      (strip(S, TimeRmTr),
      stablefailure(P, failure(TimeRmTr, X)))).
```

4.3. Additional Checking

In reality, most processes are non-terminating, so it would not be possible to retrieve all possible traces of a process. However, by given a specific trace of a trace fragment, we are able to identify whether it is a legal trace for a specific process. For instance, the following clause is used to query if a sequence of event is a trace of the system, where P is a process expression and X is a sequence of events.

```
trace(P,X) :- superstep(P,X,_).
```

In addition to proving pre-specified assertions, one distinguished feature of our approach is that implicit assertions may be proved. For example, we may identify the lower or upper bound of a (time or data) variable, which is very useful for applications like worst or best case analysis of execution time.

```
dur(P, Q, T1,T2) :- tos(P,T1,_,Q,T2).
dur(P, Q,T1,T2)
    :- tos(P,T1,_,P1,T3),dur(P1,Q,T3,T2).
```

We are able to compute the duration of the execution of one process P to its subsequent process Q by the above two rules, where T_1 is the starting time and T_2 is the ending time. By using the predicate *dur*, we are able to get identify the

```
proc(trainp1,delay(entercrossing,eventprefix(leavecrossing,eventprefix(outind,train)),20)).
proc(trainp,eventprefix(trainnear, delay(nearind,P,300))):-proc(trainp1,P).
proc(train, recursion([train, P],P)):-proc(trainp,P).
proc(pchoice1,delay(downcom,eventprefix(down,eventprefix(confirm,gate)),100)).
proc(pchoice2,delay(upcom,eventprefix(up,eventprefix(confirm,gate)),100)).
proc(pchoices, extchoice(P1,P2)):-proc(pchoice1,P1),proc(pchoice2,P2).
proc(gate, recursion([gate,P], P)):-proc(pchoices,P).
proc(cchoice1,delay(nearind,eventprefix(downcom,eventprefix(confirm,controller)),1)).
proc(cchoice2,delay(outind,eventprefix(upcom,eventprefix(confirm,controller)),1)).
proc(cchoices,extchoice(C1,C2)):-proc(cchoice1,C1),proc(cchoice2,C2).
proc(controller,recursion([controller,C],C)):- proc(cchoices,C).
proc(crossing, parallel(C,G,A,B)):-proc(controller,C),proc(gate,G),sigma1(C,A),sigma1(G,B).
proc(system, parallel(T,C,A,B)):-proc(crossing,C),proc(train,T),sigma1(T,A),sigma1(C,B).
```

Figure 3. Railway Crossing Model in CLP

lower bound of some processes involving time. The process $WAIT(2);\ a \xrightarrow{3} SKIP$ should terminate in more than 5 time units, which can be identified by the following goal and expecting T≥5.

$$?- dur(sequ(wait(2), delay(a, skip, 3)), stop, 0, T).$$

An additional property that is beyond the capability of traditional model checking is symmetry. We allow the proving of the symmetry of the system, which may be subsequently used to optimize any proving over the system (refer to [12]).

5. A Case Study: The Railway Crossing

We use a simple model of a railway crossing as a case study, which is nevertheless complex enough to demonstrate a number of aspects of the modelling and verification of timed systems. The verification of timewise refinement is also presented.

The system consists of three components: a train, a gate and a gate controller. The gate should be up to allow traffic to pass when no train is approaching, but should be lowered to obstruct traffic when a train is reaching the crossing. It is the task of the controller to monitor the approach of a train, and to instruct the gate to be lowered within the appropriate time. The train is modelled at a high level of abstraction: the only relevant aspects of the train's behavior are when it is near the crossing, when it is entering it, when it is leaving it; and the minimum delays between these events.

The gate controller *CONTROLLER* receives two types of signal from the crossing sensors: *nearind*, which informs the controller that the train is approaching, and *outind*, which indicates that the train has left the crossing. It sends two types of signal to the crossing gate mechanism: *downcom*, and *upcom*, which instruct the gate to go down and up respectively. It also receives a confirmation *confirm* from the gate. These five events form the alphabet *C* of the controller.

The gate, modelled by *GATE*, responds to the commands sent by the controller. The additional events up and down are included to model the position of the gate. These two events together with *upcom* and *downcom* and the confirmation *confirm*, form the alphabet *G* of the gate.

The train triggers the sensors by means of the *nearind* and *outind* events. The events *trainnear*, *entercrossing* and *leavecrossing* model respectively the situations where the train is close to the crossing, the train enters the crossing, and the train leaves the crossing. These five events are all that are required for the sake of this analysis: they form the alphabet *T* of the train.

The Timed CSP model of this system is as follows (presented in [16]) and the corresponding CLP model is presented in Figure 3:

$$\begin{aligned}
TRAIN &= \mu T \bullet trainnear \rightarrow nearind \xrightarrow{300} entercrossing \\
&\quad \xrightarrow{20} leavecrossing \rightarrow outind \rightarrow T \\
GATE &= \mu G \bullet downcom \xrightarrow{100} down \rightarrow confirm \rightarrow G \\
&\quad \Box\ upcom \xrightarrow{100} up \rightarrow confirm \rightarrow G \\
CONTROLLER &= \mu C \bullet outind \xrightarrow{1} upcom \rightarrow confirm \rightarrow C \\
&\quad \Box\ nearind \xrightarrow{1} downcom \rightarrow confirm \rightarrow C \\
CROSSING &= CONTROLLER\ {}_C||_G\ GATE \\
SYSTEM &= TRAIN\ {}_T||_{C \cup G}\ CROSSING
\end{aligned}$$

The time information we have about the system is that: the train takes at least 5 minutes from triggering the near.ind sensor to reach the crossing; and that it takes at least 20 seconds to get across the crossing. The controller takes a negligible amount of time, say 1 second, from receiving a signal from a sensor to relaying the corresponding instruction to the gate. The gate process takes a non-negligible amount of time, i.e. 100 seconds to get the gate into position following an instruction.

In the verification experiment of the railway crossing example, we designed and verified a number of properties, including deadlock-free property, timing properties and trace-based properties, six of which are illustrated in Table 1. As UPPAAL is by far the most well-known tool for verification of real-time systems, we did compared our approach with UPPAAL in terms of both efficiency and supported properties. The conclusion is that for properties that can be

Property	Goal in CLP
deadlock-free	proc(system, P), tdeadlock(P, 0) $\models$ *false*
if train enters crossing, the gate must be down	proc(system, P), supersetp(P, X), last(X, entercrossing),filter(X, [up,down], X2), last(X2, up) $\models$ *false*
lower bound for a train passes the crossing is 320 s	proc(system, P), dur(delay(nearind, _, _), eventprefix(outind, _), T1, T2),T2-T1<320 $\models$ *false*
if the gate is up, the train must have left the crossing	proc(system,P), superstep(P, X), not(not in([up, entercrossing, leavecrossing], X);after(X, leavecrossing, entercrossing))) $\models$ *false*
legal trace checking lower bound for a train	proc(system, P), superstep(P, [trainnear, nearind, downcomm, down, confirm, entercrossing, leavecrossing, outind])$\models$ *true*
lower bound for the controller senses train out till gate confirmed up is 101s	proc(system,P), superstep(P,X,), duration(X,outind,confirm,T),T<101 $\models$ *false*

Table 1. Properties verification

checked by both UPPAAL and our work, our tool performs similarly timewizely comparing to UPPAAL. The characteristic of our work is that properties beyond the capability of UPPAAL like timewise refinement, traced-based properties, lower or upper bound of a time internal can be effectively checked.

6. Conclusions

In this paper, we propose a reasoning tool for Timed CSP based on constraint logic, which to our knowledge, is the first mechanical reasoning support for Timed CSP. The contribution of this work is threefold. Firstly we show that event-based process algebra Timed CSP can be encoded in CLP by encoding both the operational semantics and denotational semantics. Our work therefore broadened real-time systems which can be specified and verified by CLP. Second, we investigated a wide range of properties that may be proved based on constraint solving, for instance we showed that using a unique interpretation, traditional safety and liveness can be proved effectively as well as properties such as lower or upper bound of a variable and refinement. Lastly, we implemented a prototype program and applied our approach to various systems.

Acknowledgement

The authors thank Andrew Santosa for insightful discussion on CLP and pointing out relevant documentations.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[2] P. J. Brooke. *A Timed Semantics for a Hierarchical Design Notation*. PhD thesis, University of York, april 1999.

[3] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/Event-based Software Model Checking. In *Proceeding of Integrate Formal Methods 2004*, pages 128–147, April 2004.

[4] J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.

[5] J. Davies and S. Schneider. A brief history of timed csp. *Theor. Comput. Sci.*, 138(2):243–271, 1995.

[6] Formal Systems (Europe) Ltd. Failure divergence refinement: Fdr2 user manual. 1997.

[7] G. Gupta and E. Pontelli. A constraint-based approach for specification and verification of real-time systems. In *IEEE Real-Time Systems Symposium*, pages 230–239, 1997.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[9] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[10] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The clp(r) language and system. *ACM Trans. Program. Lang. Syst.*, 14(3):339–395, 1992.

[11] J. Jaffar, A. Santosa, and R. Voicu. Modeling systems in clp with coinductive tabling. In *ICLP*, 2005.

[12] J. Jaffar, A. E. Santosa, and R. Voicu. A clp proof method for timed automata. In *RTSS*, pages 175–186, 2004.

[13] B. P. Mahony and J. S. Dong. Timed communicating object z. *IEEE Trans. Software Eng.*, 26(2):150–177, 2000.

[14] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In L. Kott, editor, *ICALP*, volume 226 of *Lecture Notes in Computer Science*, pages 314–323. Springer, 1986.

[15] A. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[16] S. Schneider. *Concurrent and Real-time System: The CSP approach*. JOHN WILEY & SONS, LTD, 2000.

[17] S. A. Schneider. An operational semantics for timed CSP. In *Proceedings Chalmers Workshop on Concurrency, 1991*, pages 428–456. Report PMG-R63, Chalmers University of Technology and University of Göteborg, 1992.

[18] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems-an integration of object-z and csp. *Formal Methods in System Design*, 18(3):249–284, 2001.

[19] D. S. Warren. Programming with tabling in XSB. In *PROCOMET '98: Proceedings of the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods*, pages 5–6, London, UK, UK, 1998.

[20] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall International, 1996.