A Reasoning Method 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 construct a reasoning method using Constraint Logic Programming (CLP) as an underlying reasoning mechanism 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. Counter-examples are generated when properties are not satisfied. Moreover, our method also handles useful extensions to Timed CSP. Finally, we demonstrate the effectiveness of our approach through case study of standard real time systems.

1 Introduction

Event-based specification languages like the classic Communicating Sequential Process (CSP) of Hoare's [7] 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 range of systems, including communication protocols, embedded systems, etc [15]. 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* mechanized verification support is its model checker FDR (Failure Divergence Refinement [5, 15]), which verifies various properties by showing that there is a refinement relation from the constructed CSP model to the CSP process capturing the properties. However, 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 than the failure semantics 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.

* Author for correspondence, phone: +65 65162834, fax: +65 67794580

Constraint Logic Programming (CLP [9]) is designed for mechanized proving based on constraint solving. CLP has been successfully applied to model programs and transition systems for the purpose of verification [6, 11], showing that their approach outperforms the well-know state-of-art systems with higher efficiency. [1] employs a logic program transformation based approach for inductive verification of real-life parameterized protocols. In this work, we propose a constraint-based approach for solving the verification problem of Timed CSP, which readily implies we handle untimed CSP as well. It is the first reasoning mechanism for Timed CSP. The challenge is to cope with the great expressiveness of Timed CSP and allow efficient automatic proving of various assertions.

Our approach starts with a systematic translation of the semantics of Timed CSP into CLP. Both operational and denotational semantics are encoded, which are used for verifying different kinds of properties. 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 specification languages, 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 investigate 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. Moreover, we are also able to generate counter examples if the properties are not satisfied. We implemented a prototype as a CLP($\mathcal{R}$) program and experimented our encoding with standard real-time systems.

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 case studies. Section 6 concludes the paper.

2 Background

2.1 Timed CSP

Hoare's CSP [7] 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:

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

$\textsc{Run}_\Sigma$ 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 (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. The external choice operator ($\Box$) allows a process of choice of behavior according to what events are requested by its environment. Internal choice represents variation in behavior determined by the internal state of the process. 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 $P_1 \,|[\, A \,]|\, P_2$). 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 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 processes which repeatedly act as $P(X)$.

Example 1 (Timed vending machine). 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.

$$
\begin{array}{l}
\mathit{TVM} \mathrel{\widehat{=}} \mu\, X \bullet \mathit{coin} \to ((\mathit{reqrelease} \to \mathit{release} \xrightarrow{2} X) \\
\qquad \Box\, (\mathit{coffee} \xrightarrow{3} \mathit{dispatchcoffee} \to X) \,\Box\, (\mathit{tea} \xrightarrow{2} \mathit{dispatchtea} \to X)) \\
\qquad \triangleright\{10\}\, (\mathit{release} \to X)
\end{array}
$$

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, and in some cases, more efficient than other kinds of programs. The CLP scheme defines a class of languages based upon the paradigm of rule-based constraint programming, where CLP($\mathcal{R}$) is an instance of this class. We present some preliminary definitions about CLP [9].

Example 2 (Factorial). The following is typical CLP program:

$fac(0, 1).$
$fac(N, X_1 * N) : -N > 0, fac(N - 1, X_1).$

A relation *fac(N, X)* is defined, where X is the factorial of N, denoted as $X = N!$. There are two atoms for the relation *fac(N, X)*, where the first atom is a *fact* and the second one is a *rule*.

The *universe of discourse* $\mathcal{D}$ of our CLP program is a set of terms, integers, and lists of integers. A *Constraint* is written using a language of functions and relations. They are used in two ways, in the basic programming language to describe expressions and conditions, and in user assertions, defined below. In this paper, we will not define the constraint language explicitly, but invent them on demand in accordance with our examples. Thus the terms of our CLP programs include the function symbols of the constraint language.

An *atom*, is as usual, of the form $p(\tilde{t})$, where p is a user defined predicate symbol and $\tilde{t}$ is a sequence of terms. A *rule* is of the form $A : -\tilde{B}, \Psi$ where the atom A is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and the constraint Ψ constitute the *body* of the rule. A *goal* has exactly the same format as the body of the rule of the form $? - \tilde{B}, \Psi$. If $\tilde{B}$ is an empty sequence of atoms, we call this a (constrained) *fact*. All goals, rules and facts are terms. A *ground instance* of a constraint, atom and rule is defined in obvious way. A *ground instance* of a constraint is obtained by instantiating variables therein from $\mathcal{D}$. The *ground instances* of a goal G, written $[\![G]\!]$ is the set of ground atoms obtained by taking all the true ground instances of G and then assembling the ground atoms therein into a set. We write $G_1 \models G_2$ to mean that for all groundings θ of G_1 and G_2, each ground atom in $G_1\theta$ appears in $G_2\theta$.

Let $G = (B_1, ..., B_n, \Psi)$ and P denote a goal and program respectively. Let $R = A : -C_1, ..., C_m, \Psi_1$ denote a rule in P, written so as none of its variables appear in G. Let $A = B$, where A and B are atoms, be shorthand for equations between their corresponding arguments. A *reduct* of G using R is of the form

$$(B_1, ..., B_{i-1}, C_1, ..., C_m, B_{i+1}, ..., B_n, B_i = A \wedge \Psi \wedge \Psi_1)$$

provided $B_i = A \wedge \Psi \wedge \Psi_1$ is satisfiable. A *derivation sequence* is a possibly infinite sequence of goals $G_0, G_1, ...$ where $G_i, i > 0$ is a reduct of G_{i-1}. If there is a last goal G_n with no atoms, notationally $(\Box, \Psi)$ and called a *terminal goal*, we say that the derivation is a *successful* and that the *answer constraint* is Ψ. A derivation is ground if every reduction therein is ground.

Example 3 (Derivation). We calculate the 3! through the goal $? - \ fac(3, \ X)$. The following demonstrates a derivation sequence of the goal with three steps. The constraints in the last step which are the termination goal answer $X = 6$.

$$N = 3,\ fac(N,\ X).$$
$$\Downarrow$$
$$N = 3, N > 0, N - 1 = N_1, X = N * X_1, fac(N_1, X_1).$$
$$\Downarrow$$
$$N = 3, N > 0, N - 1 = N_1, X = N * X_1,$$
$$N_1 > 0, N_1 - 1 = N_2, X1 = N_1 * X_2, fac(N_2, X_2).$$
$$\Downarrow$$
$$N = 3, N > 0, N - 1 = N_1, X = N * X_1, N_1 > 0, N_1 - 1 = N_2,$$
$$X1 = N_1 * X_2, N_2 > 0, N_2 - 1 = 0, X_2 = 1.$$

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. 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.

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. A relation of the form $proc(N, P)$ is used to present a process P with name N. For instance, Figure 1 is the syntax encoding of process TVM in CLP, which is a recursive process with name tvm.

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

Fig. 1. Timed Vending Machine 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 denote the *t*imed *o*perational *s*emantics, by capturing both evolution relations and timed event transition relations. 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 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. Therefore, if a process is irregular (i.e. its trace is irregular as in automata theory), 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 primitive process expressions in Timed CSP are defined through the following clauses:

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

The only transition for process STOP is time elapsing. Process SKIP may choose to wait some time before engaging event *termination* which is our choice of representation for event ✓ in CLP. Process RUN may either let time pass or engage 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 the alphabetized parallel composition operator $P_1 \ {}_X||_Y \ P_2$ are illustrated as the following [17]:

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

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

[1] Or both at the same time by engaging an nontrivial action which takes time (necessary for only extensions to Timed CSP like TCOZ [12] where E could be a complicated computation)

$tos(eventprefix(E, P), T1, [], eventprefix(E, P), T1 + D) : -D > 0.$
$tos(eventprefix(E, P), T, [E], P, T).$
$tos(prefixchoice(X, P), T, [Y], P, T) : -member(Y, X).$
$tos(prefixchoice(\_, P), T1, [], P, T1 + D) : -D > 0.$
$tos(timeout(Q1, \_, \ ), T, [E], P, T) : -tos(Q1, T, [E], P, T).$
$tos(timeout(\ , Q2, D), T, [tau], Q2, T) : -D = 0.$
$tos(timeout(Q1, Q2, D), T, [tau], timeout(P, Q2, D), T)$
$\quad : -tos(Q1, T, [tau], P, T).$
$tos(timeout(Q1, Q2, D), T1, [], timeout(P, Q2, D - T), T1 + T)$
$\quad : -T > 0, T <= D, tos(Q1, T1, [], P, T1 + T).$
$tos(wait(D), T1, E, P, T2) : -tos(timeout(stop, skip, D), T1, E, P, T2).$
$tos(extchoice(P1, \ ) , T, [E], P3, T) : -tos(P1, T, [E], P3, T).$
$tos(extchoice(\ , P2), T, [E], P4, T) : -tos(P2, T, [E], P4, T).$
$tos(extchoice(P1, P2), T, [tau], extchoice(P3, P2), T) : -tos(P1, T, [tau], P3, T).$
$tos(extchoice(P1, P2), T, [tau], extchoice(P1, P4), T) : -tos(P2, T, [tau], P4, T).$
$tos(extchoice(P1, P2), T1, [], extchoice(P3, P4), T2)$
$\quad : -T2 > T1, tos(P1, T1, [], P3, T2), tos(P2, T1, [], P4, T2).$
$tos(interleave(P1, P2), T, E, interleave(P3, P2), T)$
$\quad : -tos(P1, T, E, P3, T), (E == []; \ E == [tau]).$
$tos(interleave(P1, P2), T, E, interleave(P1, P4), T)$
$\quad : -tos(P2, T, E, P4, T), (E == []; \ E == [tau]).$
$tos(interleave(P1, P2), T, [E], interleave(P3, P2), T) : -tos(P1, T, [E], P3, T).$
$tos(interleave(P1, P2), T, [E], interleave(P1, P3), T) : -tos(P2, T, [E], P3, T).$
$tos(interleave(P1, P2), T1, [], interleave(P3, P4), T1 + D)$
$\quad : -D > 0, tos(P1, T1, [], P3, T1 + D), tos(P2, T1, [], P4, T1 + D).$
$tos(interleave(P1, P2), T, [termination], interleave(P3, P4), T)$
$\quad : -tos(P1, T, [termination], P3, T), tos(P2, T, [termination], P4, T).$
$tos(hiding(P1, X), T, [tau], hiding(P2, X), T)$
$\quad : -tos(P1, T, [E], T, P2), member(E, X).$
$tos(hiding(P1, X), T, [E], hiding(P2, X), T)$
$\quad : -tos(P1, T, [E], P2, T), not(member(E, X)).$
$tos(hiding(P1, X), T1, [], hiding(P2, X), T1 + D)$
$\quad : -D > 0, tos(P1, T1, [], P2, T1 + D),$
$\qquad not(member(A, X), tos(P1, \_, [A], \_, \_)).$
$tos(sequential(P1, P2), T, [E], sequential(P3, P2), T)$
$\quad : -tos(P1, T, [E], P3, T), not(E = termination).$
$tos(sequential(P1, P2), T, [termination], P2, T) : -tos(P1, T, [termination], \ , T).$
$tos(sequential(P1, P2), T1, [], sequential(P3, P2), T1 + D)$
$\quad : -D > 0, tos(P1, T1, [], P3, T1 + D), not(tos(P1, \_, [termination], \_, \_)).$
$tos(interrupt(P1, P2), T, [E], interrupt(P3, P2), T) : -tos(P1, T, [E], P3, T).$
$tos(interrupt(\_, P2), T, [E], P3, T) : -tos(P2, T, [E], P3, T).$
$tos(interrupt(P1, P2), T1, [], interrupt(P3, P4), T1 + D)$
$\quad : -D > 0, tos(P1, T1, [], P3, T1 + D), tos(P2, T1, [], P4, T1 + D).$

Fig. 2. Operational Semantics of Timed CSP in CLP

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

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

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

$$\begin{array}{l} tos(para(P1, P2, X, Y), T, [E], para(P3, P2, X, Y), T) \\ \quad :- tos(P1, T, [E], P3, T), member(E, X), not(member(E, Y)). \\ tos(para(P1, P2, X, Y), T, [E], para(P1, P4, X, Y), T) \\ \quad :- os(P2, T, [E], P4, T), member(E, Y), not(member(E, X)). \\ tos(para(P1, P2, X, Y), T, [E], para(P3, P4, X, Y), T) \\ \quad :- tos(P1, T, E, P3, T), tos(P2, T, E, P4, T), \\ \qquad member(E, X), member(E, Y). \\ tos(para(P1, P2, X, Y), T1, [], para(P3, P4, X, Y), T1 + D) \\ \quad :- tos(P1, T1, [], P3, T1 + D), tos(P2, T1, [], P4, T1 + D). \end{array}$$

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 our rules and the operators which are partly illustrated in Figure 2 and fully at our website[2]. Therefore, the soundness of the encoding can be proved by showing there is a bi-simulation relationship [13] between the transition system interpretation defined in [17] and ours, and 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 be 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., $P(X)$.

$$\begin{array}{l} tos(recursion([N, P], P1), T, [E], recursion([N, P], P2), T) \\ \quad :- not(P1 == N), tos(P1, T, [E], P2, T). \\ tos(recursion([N, P], P1), T1, [], recursion([N, P], P2), T1 + D) \\ \quad :- D > 0, tos(P1, T1, [], P2, T1 + D). \\ tos(recursion([N, P], N), T, [], recursion([N, P], P), T). \end{array}$$

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

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]. A timed failure is a record of an execution, consisting of a timed trace which contains information about event performed, and a timed refusal which contains information about when events could be refused. 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,

$$
\begin{array}{l}
timedfailure(stop, failure([], \_)).\\
timedfailure(skip, failure([], R))\\
\quad :- sigma(R, S), not(member(termination, S)).\\
timedfailure(skip, failure([tevent(T, termination)], R))\\
\quad :- T >= 0, before(R, T, Z), sigma(Z, N), not\ member(termination, N).
\end{array}
$$

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. 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.

$$
\begin{array}{l}
timedfailure(parallel(Q1, Q2, A), failure(S, N))\\
\quad :- timedfailure(Q1, failure(S1, N1)),\\
\qquad timedfailure(Q2, failure(S2, N2)), union(N1, N2, N),\\
\qquad union(A, [termination], AT), remove(N1, AT, N11),\\
\qquad remove(N2, AT, N22), setequal(N11, N22), tsynch(S1, S2, A, S).
\end{array}
$$

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 (formal definition in [16]). The interface parallel operator requires synchronization on events from the interface event set A, and interleaving on events not in A.

Notice that 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.

3.3 Handling Extensions to Timed CSP

Timed CSP is introduced in [14]. Since then, various extensions of Timed CSP have been proposed. In this work, we identify some of the effective extensions 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. For any observation that can be extended into the future, the only events that 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.

$$
\begin{array}{l}
sigTF(eventprefix(E,\_,\_), sigfailure([], X, T)) \\
\quad :- not(E == sig(\_)), sigma(X, Z), \\
\qquad not(member(E, Z)), end(X, T1), T >= T1. \\
sigTF(eventprefix(E, P, D), sigfailure([tevent(T, E) \mid XS], Y, T1 + D + T)) \\
\quad :- T >= 0, not(E == sig(\_)), sigTF(P, sigfailure(S, Y1), T1), \\
\qquad backthrough(Y, T + D, Y1), begin(S, T2), T2 >= T + D, \\
\qquad end(S, Y, T3), max(T, T3, T4), T1 + D + T >= T4, \\
\qquad before(Y, T, Z), sigma(Z, N), \\
\qquad not(member(E, N)), delay(S, T + D, XS). \\
sigTF(eventprefix(sig(E), P, D), sigfailure([], [], 0)). \\
sigTF(eventprefix(sig(E), P, D), sigfailure([tevent(0, E) \mid XS], Y, T)) \\
\quad :- sigTF(P, failure(S, Y1), T1), backthrough(Y, T + D, Y1), \\
\qquad T = T1 + D, before(Y, T, Z), sigma(Z, N), \\
\qquad not(member(E, N)), delay(S, T + D, XS).
\end{array}
$$

The 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 CLP clauses illustrate the possible evolution of signal event prefixing. 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 occurrence 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 [12, 18]. Instead of adopting a heavy

language like TCOZ (Timed Communicating Object-Z [12]), we allow a finite number of variables to be associated with a process[3], 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 this vending machine, in which we allow different coins to be inserted via a channel communication $coin?x$ where x is 10, 20 or 50, a data variable $Quota$ is requested to accumulate the amount of all coins inserted by the user.

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

where $AddQuota$ is an operation defined in Z, which is:

$$AddQuota \mathrel{\widehat{=}} [x?, quota, quota' : \mathbb{N} \mid quota' = quota + x?]$$

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 $Quota2$, 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 $Quota$.

$$proc(coin, eventpreifx(coin(X1), addquota), Quota1, Quota2)$$
$$\quad :- action(addquota, X1, Quota1, Quota2).$$
$$action(addquota, X1, Quota1, Quota2) :- Quota2 = Quota1 + X1.$$

4 Proving Properties of Timed CSP

This section is devoted to 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($\mathcal{R}$). 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 [8].

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

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[4]:

$$treachable(P, P, [], T1, T1).$$
$$treachable(P, Q, [E \mid N], T1, T2)$$
$$\quad :- tos(P, T1, E, P1, T3), treachable(P1, Q, N, T3, T2).$$

The relation *treachable(P, Q, N, T1, T2)* states that it is possible to reach the process expression Q at time *T2* from P at time T1, with trace N. 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:

$$inv(P, T, Property) :- not(treachable(P, Q, \_, T, T1), not\ sat(Property)).$$

where *not sat(Property)* is a constraint indicating that the output from the previous atom not satisfying the user defined *Property*.

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

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

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 trace as a counterexample. Alternatively, we may present it as a result of the deadlock proving.

We allow trace-based properties (safety or liveness) 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:

$$superstep(P, [\_], \_) :- not(tos(P, \_, \_, Q, \_), not\ table(Q)).$$
$$superstep(P, [A \mid N], Q) :- tos(P, \_, M, P1, \_), not(M == []; M == [tau]),$$
$$\quad M = [A], not\ table(P1), assert(table(P1)), superstep(P1, N, Q).$$
$$superstep(P, N, Q) :- tos(P, \_, M, P1, \_), (M == []; M == [tau]),$$
$$\quad not\ table(P1), assert(table(P1)), superstep(P1, N, Q).$$

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.

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

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

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*.

Example 4 (Verification). 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 checked by FDR [15], timedwise refinements for Timed CSP contain more information 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 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:

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

Detailed explanation can be found in [16]. 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 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 is 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 are allowed by the stable failures of P. It is formally defined as in [16]:

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

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, which 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 an event sequencing of a given 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 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.

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

5 Experiments and Results

In this section, we compare our method to the mature model checker for CSP, namely FDR (version 2.78), in terms of flexibility as well as efficiency. We implement a prototype as a normal CLP($\mathcal{R}$) program. In the following, we demonstrate our experiments with three examples on a Unix system located at a Sunfire sever with IGB user memory. Because FDR is designed for CSP, the quantitative timing aspects of the examples have been abstracted before FDR verification.

Timed Vending Machine The specification of the timed vending machine is presented in Example 1. Figure 1 shows the timed vending machine model in CLP. This example is customized into a FDR program (say P), in which the time-out operator is replaced with an external choice. The following are the properties verified:

- *tvm-1* Deadlock-freeness
- *tvm-2* Trace timewise refinement:
  - in CLP, whether the process TVM is a trace timewise refinement of P.
  - in FDR, whether the process P is a trace refinement of TVM.
- *tvm-3* Whether there is such a case that coffee is selected while tea is dispatched.

Dining Philosopher The classic dining philosopher example is also experimented. The specification is available in [7]. We implemented this example with N philosophers and N forks. The following properties are experimented:

- *philosopherN-1* It is not deadlock-free
- *philosopherN-2* No more than N+1/2 philosophers can eat at the same time.
- *philosopherN-3* It is possible that one philosopher eat all the time with the others starving. This property is checked with trace refinement.

The Railway Crossing The railway crossing system is modelled and checked, which is complex enough to demonstrate a number of aspects of the modelling and verification of timed systems. The system consists of three components: a train, a gate and a controller. The gate should be up to allow traffic to pass

Property	Goal in CLP
deadlock-freeness	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 320s	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	proc(system, P),superstep(P, [trainnear, nearind, downcomm,down, confirm, entercrossing, leavecrossing, outind])$\models$*true*

Table 1. Properties Verification

when no train approaching and lowered to obstruct traffic when a train is coming. The controller monitors the approach of a train, and instructs the gate to be lowered within the appropriate time. The train is modelled abstractly with behaviors: nearing, entering and leaving the crossing. The Timed CSP modelling is as follows (originally presented in [16]):

$$
\begin{aligned}
&\mathit{TRAIN} \mathrel{\widehat{=}} \mu\, T \bullet \mathit{trainnear} \rightarrow \mathit{nearind} \stackrel{300}{\rightarrow} \mathit{entercrossing} \\
&\qquad \stackrel{20}{\rightarrow} \mathit{leavecrossing} \rightarrow \mathit{outind} \rightarrow T \\
&\mathit{GATE} \mathrel{\widehat{=}} \mu\, G \bullet \mathit{downcom} \stackrel{100}{\rightarrow} \mathit{down} \rightarrow \mathit{confirm} \rightarrow G \\
&\qquad \Box\ \mathit{upcom} \stackrel{100}{\rightarrow} \mathit{up} \rightarrow \mathit{confirm} \rightarrow G \\
&\mathit{CONTROLLER} \mathrel{\widehat{=}} \mu\, C \bullet \mathit{outind} \stackrel{1}{\rightarrow} \mathit{upcom} \rightarrow \mathit{confirm} \rightarrow C \\
&\qquad \Box\ \mathit{nearind} \stackrel{1}{\rightarrow} \mathit{downcom} \rightarrow \mathit{confirm} \rightarrow C \\
&\mathit{CROSSING} \mathrel{\widehat{=}} \mathit{CONTROLLER}\ {}_{C}||_{G}\ \mathit{GATE} \\
&\mathit{SYSTEM} \mathrel{\widehat{=}} \mathit{TRAIN}\ {}_{T}||_{C \cup G}\ \mathit{CROSSING}
\end{aligned}
$$

The time information of the system is that: the train takes at least 5 minutes from triggering the near.ind sensor to reach the crossing; and 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 100 seconds to get itself into position following an instruction. A number of interesting properties can be formulated, evidenced in Table 1. The three properties selected for comparing our approach with FDR verification are:

- *railway-1* Deadlock-freeness
- *railway-2* Whether trace $\langle \mathit{trainnear}, \mathit{nearind}, \mathit{downcomm}, \mathit{down}, \mathit{confirm}, \mathit{entercrossing}, \mathit{leavecrossing}, \mathit{outind} \rangle$ is a legal trace or not.
- *railway-3* Whether the lower bound for a train passes the crossing is $320s$.

Assertion	CLP($\mathcal{R}$) *(sec)*	FDR *(sec)*	Assertion	CLP($\mathcal{R}$) *(sec)*	FDR *(sec)*
tvm-1	0.00	0.23	*phi3-1*	0.12	0.25
tvm-2	0.03	0.27	*phi3-2*	0.22	–
tvm-3	0.01	–	*phi3-3*	0.04	0.17
railway-1	0.25	0.25	*phi4-1*	0.84	0.28
railway-2	0.02	0.26	*phi4-2*	2.5	–
railway-3	0.32	–	*phi4-3*	0.1	0.3

Table 2. Experiment Results

We summarize our results in Table 2. We ran the examples in both CLP($\mathcal{R}$) and FDR systems and we calculated the execution time of each property if the property is able to be checked in that system. From the table, we can see that most of our timing analysis performance are competitive with the well-known system, while in some cases, we are not so competitive. The important metric of our experiments is the flexibility. The results show that our reasoning method based on constraint solver can handle a wider range of properties, including the timed-related properties, bounds of variables, event specified properties, and etc.

6 Conclusions

In this paper, we proposed a reasoning method for Timed CSP based on constraint logic, which to our knowledge, is the first mechanized reasoning support for Timed CSP. The contribution of this work is fourfold. Firstly we showed that event-based process algebra Timed CSP can be encoded in CLP by encoding both the operational and denotational semantics. Our work therefore broadened real-time systems which can be specified and verified by CLP. Secondly, we handled some useful extensions to Timed CSP, most significant one is the concept of *singal* for specifying broadcast communication. Thirdly, 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. In our future work, we plan to build a graphical user interface for automatically translating Timed CSP models, inserting properties, visualizing counterexamples if any and etc, which has been partially done recently. Besides, we would also extend our method to verify other integrated formalisms which are based on CSP/Timed CSP.

Acknowledgement

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

References

1. R. Abhik and I.V. Ramakrishnan. Automated Inductive Verification of Parameterized Protocols. In *International Conf. on Computer Aided Verification (CAV)*. Springer, 2001.
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, 2004.
4. J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.
5. Formal Systems (Europe) Ltd. Failure Divergence Refinement: FDR2 User Manual. 1997.
6. G.l 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.
7. C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.
8. A. Santosa J. Jaffar and R. Voicu. Modeling Systems in CLP with Coinductive Tabling. In *International Conference on Logic Programming*, 2005.
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. E. Santosa, and R. Voicu. A CLP Proof Method for Timed Automata. In *Real-Time Systems Symposium*, pages 175–186, 2004.
12. B. P. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Trans. Software Eng.*, 26(2):150–177, 2000.
13. R. Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, 1980.
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. W. 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, 1998.
20. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall International, 1996.