Declarative Diagnosis of Temporal Concurrent Constraint Programs

M. Falaschi[1], C. Olarte[2,4], C. Palamidessi[2], and F. Valencia[3]

[1] Dip. Scienze Matematiche e Informatiche, Università di Siena, Italy.
moreno.falaschi@unisi.it.
[2] INRIA Futurs and LIX, École Polytechnique, France.
[3] CNRS and LIX, École Polytechnique, France.
{colarte,catuscia, fvalenci}@lix.polytechnique.fr.
[4] Dept. Computer Science, Javeriana University, Cali, Colombia.

Abstract. We present a framework for the declarative diagnosis of nondeterministic timed concurrent constraint programs. We present a denotational semantics based on a (continuous) immediate consequence operator, $T_{\mathcal{D}}$, which models the process behaviour associated with a program $\mathcal{D}$ given in terms of sequences of constraints. Then, we show that, given the intended specification of $\mathcal{D}$, it is possible to check the correctness of $\mathcal{D}$ by a single step of $T_{\mathcal{D}}$. In order to develop an effective debugging method, we approximate the *denotational semantics* of $\mathcal{D}$. We formalize this method by abstract interpretation techniques, and we derive a finitely terminating abstract diagnosis method, which can be used statically. We define an abstract domain which allows us to approximate the infinite sequences by a finite 'cut'. As a further development we show how to use a specific linear temporal logic for deriving automatically the debugging sequences. Our debugging framework does not require the user to either provide error symptoms in advance or answer questions concerning program correctness. Our method is compositional, that may allow to master the complexity of the debugging methodology.

Keywords: timed concurrent constraint programs, (modular) declarative debugging, denotational semantics, specification logic.

1 Introduction

The main motivation for this work is to provide a methodology for developing effective (modular) debugging tools for timed concurrent constraint (`tcc`) languages.

Finding program bugs is a long-standing problem in software construction. However, current debugging tools for `tcc` do not enforce program correctness adequately as they do not provide means to find bugs in the source code w.r.t. the intended program semantics. We are not aware of the existence of sophisticated debuggers for this class of languages. It would be certainly possible to define some trace debuggers based on suitable extended box models which help display the execution. However, due to the complexity of the semantics of `tcc`

programs, the information obtained by tracing the execution would be difficult to understand. Several debuggers based on tracing have been defined for different declarative programming languages. For instance [11] in the context of integrated languages. To improve understandability, a graphic debugger for the multi–paradigm concurrent language Curry is provided within the graphical environment CIDER [12] which visualizes the evaluation of expressions and is based on tracing. TeaBag [3] is both a tracer and a runtime debugger provided as an accessory of a Curry virtual machine which handles non–deterministic programs. For Mercury, a visual debugging environment is ViMer [5], which borrows techniques from standard tracers, such as the use of spypoints. In [14], the functional logic programming language NUE-Prolog has a more declarative, algorithmic debugger which uses the declarative semantics of the program and works in the style proposed by Shapiro [19]. Thus, an oracle (typically the user) has to provide the debugger with error symptoms, and has to correctly answer oracle questions driven by proof trees aimed at locating the actual source of errors. Unfortunately, when debugging real code, the questions are often textually large and may be difficult to answer.

Abstract diagnosis [8] is a declarative debugging framework which extends the methodology in [10,19], based on using the immediate consequence operator to identify bugs in logic programs, to diagnosis w.r.t. computed answers. An important advantage of this framework is to be goal independent and not to require the determination of symptoms in advance. In [2], the declarative diagnosis methodology of [8] was generalized to the debugging of functional logic programs, and in [1] it was generalized to functional programs.

In this paper we aim at defining a framework for the abstract diagnosis of `tcc` programs. The idea for abstract debugging follows the methodology in [1], but we have to deal with the extra complexity derived from having constraints, concurrency, and time issues. Moreover, our framework introduces several novelties, since we delevop a compositional semantics and we show how to derive automatically the debugging symptoms from a specification given in terms of a linear temporal logic. We proceed as follows.

First, we associate a (continuous) denotational semantics to our programs. This leads us to a fixpoint characterization of the semantics of `tcc` programs. Then we show that, given the intended specification $\mathcal{I}$ of a program $\mathcal{D}$, we can check the correctness of $\mathcal{D}$ by a single step of this operator. The specification $\mathcal{I}$ may be partial or complete, and can be expressed in several ways: for instance, by (another) `tcc` program, by an assertion language [6] or by sets of constraint sequences (in the case when it is finite). The diagnosis is based on the detection of *incorrect rules* and *uncovered constraint sequences*, which both have a bottom-up definition (in terms of one application of the continuous operator to the abstract specification). It is worth noting that no fixpoint computation is required, since the (concrete) semantics does not need to be computed.

In order to provide a practical methodology, we also present an effective debugging methodology which is based on abstract interpretation. Following an idea inspired in [4,8], and developed in [1] we use *over* and *under* specifications

$\mathcal{I}^+$ and $\mathcal{I}^-$ to correctly over- (resp. under-) approximate the intended specification $\mathcal{I}$ of the semantics. We then use these two sets respectively for approximating the input and the output of the operator associated by the denotational semantics to a given program, and by a simple static test we can determine whether some of the program rules are wrong. The method is sound in the sense that each error which is found by using $\mathcal{I}^+, \mathcal{I}^-$ is really a bug w.r.t. $\mathcal{I}$.

The rest of the paper is organized as follows. Section 2 recalls the syntax of the `ntcc` calculus, a non-deterministic extension of `tcc` model. Section 3 and 4 are devoted to present a denotational semantics for `ntcc` programs. We also formulate an operational semantics and show the correspondence with the least fixpoint semantics. Section 5 provides an abstract semantics which correctly approximates the fixpoint semantics of $\mathcal{D}$. In Section 6 we introduce the general notions of incorrectness and insufficiency symptoms. We give an example of a possible abstract domain, by considering a 'cut' to finite depth of the constraint sequences and show how it can be used to detect errors in some benchmark programs. We also show how to derive automatically the elements (sequences) of the domain for debugging from a specification in linear temporal logic. Section 7 concludes.

2 The language and the semantic framework

In this section we describe the `ntcc` calculus [15], a non-deterministic temporal extension of the concurrent constraint (cc) model [18].

2.1 Constraint Systems.

cc languages are parametrized by a *constraint system.* A constraint system provides a signature from which syntactically denotable objects called *constraints* can be constructed, and an entailment relation $\models$ specifying interdependencies between such constraints.

Definition 1. ***(Constraint System).*** *A* constraint system *is a pair* (Σ, Δ) *where* Σ *is a signature specifying constants, functions and predicate symbols, and* Δ *is a consistent first-order theory over* Σ*.*

Given a constraint system (Σ, Δ), let $\mathcal{L}$ be the underlying first-order language $(\Sigma, \mathcal{V}, \mathcal{S})$, where $\mathcal{V}$ is a countable set of variables and $\mathcal{S}$ is the set of logical symbols including $\wedge$, $\vee$, $\Rightarrow$, $\exists$, $\forall$, `true` and `false` which denote logical conjunction, disjunction, implication, existential and universal quantification, and the always true and false predicates, respectively. *Constraints,* denoted by $c, d, \ldots$ are first-order formulae over $\mathcal{L}$. We say that c *entails* d in Δ, written $c \models d$, if the formula $c \Rightarrow d$ holds in all models of Δ. As usual, we shall require $\models$ to be decidable.

We say that c is equivalent to d, written $c \approx d$, iff $c \models d$ and $d \models c$. Henceforth, $\mathcal{C}$ is the set of constraints modulo $\approx$ in (Σ, Δ).

2.2 Process Syntax

In ntcc time is conceptually divided into *discrete intervals* (or *time-units*). Intuitively, in a particular time interval, a cc process P receives a stimulus (i.e. a constraint) from the environment, it executes with this stimulus as the initial store, and when it reaches its resting point, it responds to the environment with the resulting store. The resting point also determines a residual process Q, which is then executed in the next time interval.

Definition 2 (Syntax). *Processes P, Q, $\ldots \in$ Proc are built from constraints $c \in \mathcal{C}$ and variables $x \in \mathcal{V}$ in the underlying constraint system (Σ, Δ) as follows:*

$$P, Q, \ldots ::= \mathbf{skip} \mid \mathbf{tell}(c) \mid \sum_{j \in J} \mathbf{when}\ c_j\ \mathbf{do}\ P_j \mid P \parallel Q \mid (\mathbf{local}\ x)\ \mathbf{in}\, P \mid \mathbf{next}\, P \mid \star\ P \mid p(x)$$

Process **skip** does nothing. Process $\mathbf{tell}(c)$ adds the constraint c to the current store, thus making c available to other processes in the current time interval. Process $\sum_{j \in J} \mathbf{when}\ c_j\ \mathbf{do}\ P_j$ where J is a finite set of indexes, represents a process that non-deterministically choose a process P_j s.t c_j is entailed by the current store. The chosen alternative, if any, precludes the others. If no choice is possible in the current time unit, all the alternatives are precluded from execution. We shall use $\sum_{j \in J} P_j$ when the guards are `true` ("blind-choice") and we omit $\sum_{j \in J}$ when J is a singleton. Process $P \parallel Q$ represents the parallel composition of P and Q. Process $(\mathbf{local}\ x)\ \mathbf{in}\, P$ behaves like P, except that all the information on x produced by P can only be seen by P and the information on x produced by other processes cannot be seen by P.

The only move of $\mathbf{next}\, P$ is a unit-delay for the activation of P. We use $\mathbf{next}^n(P)$ as an abbreviation for $\mathbf{next}(\mathbf{next}(\ldots(\mathbf{next}\, P)\ldots))$, where **next** is repeated n times. $\star P$ represents an arbitrary long but finite delay for the activation of P. It can be viewed as $P + \mathbf{next}\, P + \mathbf{next}^2 P \ldots$.

Recursion in ntcc is defined by means of *processes definitions* of the form

$$p(x_1, .., x_n) \stackrel{\mathtt{def}}{=} A$$

$p(y_1, ..., y_n)$ is an *invocation* and intuitively the body of the process definition (i.e. A) is executed replacing the formal parameter $\boldsymbol{x}$ by the actual parameters $\boldsymbol{y}$. When $|\boldsymbol{x}| = 0$, we shall omit the parenthesis.

To avoid non-terminating sequences of internal reductions (i.e non-terminating computation within a time interval), recursive calls must be guarded in the context of **next** (see [15] for further details). In what follows $\mathcal{D}$ shall denote a set of processes definitions.

3 Operational Semantics

Operationally, the information in the current time unit is represented as a constraint $c \in \mathcal{C}$, so-called *store*. Following standard lines [18], we extend the syntax

with a construct (**local** x, d) **in** P which represents the evolution of a process of the form (**local** x) **in** Q, where d is the local information (or private store) produced during this evolution. Initially d is "empty", so we regard (**local** x) **in** P as (**local** x, `true`) **in** P.

The operational semantics will be given in terms of the reduction relations $\longrightarrow, \Longrightarrow \subseteq \mathit{Proc} \times \mathcal{C} \times \mathit{Proc} \times \mathcal{C}$ defined in Table 1. The *internal transition* $\langle P, c\rangle \longrightarrow \langle Q, d\rangle$ should be read as "P with store c reduces, in one internal step, to Q with store d ". The *observable transition* $P \stackrel{(c,d)}{\Longrightarrow} Q$ should be read as "P on input c from the environment, reduces in one time unit to Q and outputs d to the environment". Process Q is the process to be executed in the next time unit. Such a reduction is obtained from a sequence of internal reductions starting in P with initial store c and terminating in a process Q' with store d. Crudely speaking, Q is obtained by removing from Q' what was meant to be executed only during the current time interval. In `ntcc` the store d is not automatically transferred to the next time unit. If needed, information in d can be transfered to next time unit by process P.

Let us describe some of the rules for the internal transitions. Rule $TELL$ says that constraint c is added to the current store d. SUM chooses nondeterministically a process P_j for execution if its guard (c_j) can be entailed from the store. In PAR_r and PAR_l, if a process P can evolve, this evolution can take place in the presence of some other process Q running in parallel. Rule LOC is the standard rule for locality (or hiding) (see [18,9]). $CALL$ shows how a process definition is replaced by its body according to the set of process definition in $\mathcal{D}$. Finally, $STAR$ executes P in some time-unit in the future.

Let us now describe the rule for the observable transitions. Rule OBS says that an observable transition from P labeled by (c, d) is obtained by performing a terminating sequence of internal transitions from $\langle P, c\rangle$ to $\langle Q, d\rangle$, for some Q. The process to be executed in the next time interval, $F(Q)$ ("future" of Q), is obtained as follows:

Definition 3. *(**Future Function**). Let $F : \mathit{Proc} \rightharpoonup \mathit{Proc}$ be defined by*

$$F(P) = \begin{cases} \textbf{skip} & \textit{if } P = \textbf{skip} \\ \textbf{skip} & \textit{if } P = \textbf{when } c \textbf{ do } Q \\ F(P_1) \parallel F(P_2) & \textit{if } P = P_1 \parallel P_2 \\ (\textbf{local } x) \textbf{ in } F(Q) & \textit{if } P = (\textbf{local } x, c) \textbf{ in } Q \\ Q & \textit{if } P = \textbf{next } Q \end{cases}$$

In this paper we are interested in the so called *quiescent input sequences* of a process. Intuitively, those are sequences of constraints on input of which P can run without adding any information, wherefore what we observe is that the input and the output coincide. The set of quiescent sequences of a process P is thus equivalent to the observation of all sequences that P can possibly output under the influence of *arbitrary* environment. We shall refer to the set of quiescent sequences of P as the *strongest postcondition* of P written $sp(P)$, w.r.t the set of sequences of constraints denoted by $\mathcal{C}^*$. In what follows, let s, s_1,etc

$$\text{TELL}\ \frac{}{\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{skip}, d \wedge c\rangle} \qquad \text{SUM}\ \frac{d \models c_j \quad j \in J}{\left\langle \sum_{j\in J} \mathbf{when}\ c_j\ \mathbf{do}\ P_j, d\right\rangle \longrightarrow \langle P_j, d\rangle}$$

$$\text{PAR}_\text{r}\ \frac{\langle P, c\rangle \longrightarrow \langle P', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P' \parallel Q, d\rangle} \qquad \text{PAR}_\text{l}\ \frac{\langle Q, c\rangle \longrightarrow \langle Q', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P \parallel Q', d\rangle}$$

$$\text{CALL}\ \frac{p(x) :- A \in \mathcal{D}}{\langle p(x), d\rangle \longrightarrow \langle A, d\rangle} \qquad \text{STAR}\ \frac{n \geq 0}{\langle \star P, d\rangle \longrightarrow \langle \mathbf{next}^{\,n} P, d\rangle}$$

$$\text{LOC}\ \frac{\langle P, c \wedge (\exists_x d)\rangle \longrightarrow \langle P', c' \wedge (\exists_x d)\rangle}{\langle (\mathbf{local}\ x, c)\ \mathbf{in}\ P, d \wedge \exists_x c\rangle \longrightarrow \langle (\mathbf{local}\ (x, c'))\ \mathbf{in}\ P', d \wedge \exists_x c'\rangle}$$

$$\text{OBS}\ \frac{\langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow \quad R \equiv F(Q)}{P \xRightarrow{(c,d)} R}$$

Table 1. Rules for the internal reduction $\longrightarrow$ and the observable reduction $\Longrightarrow$

range over elements in $\mathcal{C}^*$ and $s(i)$ be the *i-th* element in s. We note that, as in Dijkstra's strongest postcondition approach, proving whether P satisfies a given (temporal) property A, in the presence of any environment, reduces to proving whether $sp(P)$ is included in the set of sequences satisfying A [15].

Definition 4 (Observables). *The behavioral observations that can be made of a process are given by the* strongest postcondition (or quiescent) *behavior of* P

$$sp(P) = \{s \mid P \xRightarrow{(s,s)}{}^{*} \ \textit{for some}\ s \in \mathcal{C}^*\}.$$

where $P \xRightarrow{(s,s)}{}^{*} \equiv P \xRightarrow{(s(1),s(1))} P' \xRightarrow{(s(2),s(2))} \ldots$

4 Denotational Semantics

In this section we give a denotational characterization of the strongest postcondition observables of `ntcc` following ideas developed in [15] and [17].

The denotational semantics is defined as a function $[\![\cdot]\!]$ which associates to each process a set of finite sequences of constraints, namely $[\![\cdot]\!] : (\mathit{Proc} \rightarrow \mathcal{P}(\mathcal{C}^*)) \rightarrow (\mathit{ProcHeads} \rightarrow \mathcal{P}(\mathcal{C}^*))$, where *ProcHeads* denotes the set of process names with their formal parameters. The definition of this function is given in Table 2. We use $\exists_x s$ to represent the sequence obtained by applying $\exists_x$ to each constraint in s. We call the functions in the domain $\mathit{ProcHeads} \rightarrow \mathcal{P}(\mathcal{C}^*)$ as *Interpretations*. We consider the following order on (infinite) sequences $s \leq s'$ iff $\forall i. s'(i) \models s(i)$. For what concern finite sequences we can order them similarly. Let s and s' be finite sequences, then if s has length less or equal to s' then $s \leq s'$ iff $\forall i = 1, 2, \ldots \mathit{length}(s). s'(i) \models s(i)$

Intuitively, $[\![P]\!]$ is meant to capture the quiescent sequences of a process P. For instance, all sequence whose first element is stronger than c is quiescent for $\mathbf{tell}(c)$ (D1). Process $\mathbf{next}\, P$ has not influence on the first element of a sequence, thus $d.s$ is quiescent for it if s is quiescent for P (D5). The semantics for a procedure call $p(x)$ is directly given by the interpretation I provided (D6). A sequence is quiescent for $\star P$ if there is a suffix of it which is quiescent for P (D7). The other rules can be explained analogously.

D1	$[\![\mathbf{tell}(c)]\!]_I \;=\; \{d.s \mid d \models c, s \in \mathcal{C}^*\}$
D2	$[\![\sum_{j\in J} \mathbf{when}\ c_j\ \mathbf{do}\ P_j]\!]_J \;=\; \bigcup_{j\in J}\{d.s \mid d \models c_j, d.s \in [\![P_j]\!]_J\}$ $\cup$ $\bigcap_{j\in J}\{d.s \mid d \not\models c_j, d.s \in \mathcal{C}^*\}$
D3	$[\![P \parallel Q]\!]_I \;=\; [\![P]\!]_I \cap [\![Q]\!]_I$
D4	$[\![\mathbf{local}\ x\ \mathbf{in}\ P]\!]_I \;=\; \{s \mid$ there exists $s' \in [\![P]\!]_I$ s.t. $\exists_x s = \exists_x s'\}$
D5	$[\![\mathbf{next}\ P]\!]_I \;=\; \mathcal{C}^1 \cup \{d.s \mid d \in \mathcal{C}, s \in [\![P]\!]_I\}$
D6	$[\![p(x)]\!]_I \;=\; I(p(x))$
D7	$[\![\star P]\!]_I \;=\; \{s.s' \mid s \in \mathcal{C}^*$ and $s' \in [\![P]\!]_I\}$

Table 2. Denotational semantics of `ntcc`

Formally the semantics is defined as the least fixed-point of the corresponding operator $T_{\mathcal{D}} \in (\mathit{ProcHeads} \to \mathcal{P}(\mathcal{C}^*)) \to (\mathit{ProcHeads} \to \mathcal{P}(\mathcal{C}^*))$

$$T_{\mathcal{D}}(I)(p(x)) = \quad [\![\exists_y (A \parallel d_{xy})]\!]_I \text{ if } p(y) :- A \in \mathcal{D}$$

where d_{xy} is the diagonal element used to represent parameter passing (see [18] for details). The following example illustrates the $T_{\mathcal{D}}$ operator.

Example 1. Consider a control system that must exhibit the control signal *stop* when some component malfunctions (i.e. the environment introduces as stimulus the constraint $failure$). The following (incorrect) program intends to implement such a system:

$$\mathit{control} = \mathbf{when}\ \mathit{failure}\ \mathbf{do\ next}\ \mathit{action} \parallel \mathbf{next}\ \mathit{control}$$
$$\mathit{action} = \mathbf{tell}(\mathit{stop})$$

Starting from the bottom interpretation $I_\perp$ assigning to every process the empty set (i.e $I_\perp(\mathit{control}) = I_\perp(\mathit{action}) = \emptyset$), $T_{\mathcal{D}}$ is computed as follows:

$$\begin{aligned} T_{\mathcal{D}}(I_\perp)(\mathit{control}) &= [\![\mathbf{when}\ \mathit{failure}\ \mathbf{do\ next}\ \mathit{action} \parallel \mathbf{next}\ \mathit{control}]\!]_{I_\perp} \\ &= \{d_1.d_2.s | d_1 \models \mathit{failure} \Rightarrow d_1.d_2.s \in [\![\mathbf{next}\ \mathit{action}]\!]_{I_\perp}\} \\ &\quad \cap \{\mathcal{C}^1 \cup \{d_1.s | s \in [\![\mathit{control}]\!]_{I_\perp}\}\} \\ &= \{d_1.d_2.s | d_1 \models \mathit{failure} \Rightarrow d_1.d_2.s \in \{\mathcal{C}^1 \cup \emptyset\}\} \cap \mathcal{C}^1 = \mathcal{C}^1 \\ T_{\mathcal{D}}(I_\perp)(\mathit{action}) &= [\![\mathbf{tell}(\mathit{stop})]\!]_{I_\perp} = \{d_1.s | d_1 \models \mathit{stop}\} \end{aligned}$$

Let now $I_1 = T_{\mathcal{D}}(I_\perp)$:

$$\begin{aligned} T_{\mathcal{D}}(I_1)(\mathit{control}) &= \{d_1.d_2.s | d_1 \models \mathit{failure} \Rightarrow d_1.d_2.s \in [\![\mathbf{next}\, \mathit{action}]\!]_{I_1}\} \cap \\ &\quad \big(\mathcal{C}^1 \cup \{d.s | s \in [\![\mathit{control}]\!]_{I_1}\}\big) \\ &= \{d_1.d_2.s | d_1 \models \mathit{failure} \Rightarrow d_1.d_2.s \in \big(\mathcal{C}^1 \cup \{d_1.d_2.s | d_2 \models \mathit{stop}\}\big)\} \\ &\quad \cap \big(\mathcal{C}^1 \cup \mathcal{C}^2\big) = \{d_1.d_2 | d_1 \models \mathit{failure} \Rightarrow d_2 \models \mathit{stop}\} \end{aligned}$$

5 Abstract Semantics

In this section, starting from the fixpoint semantics in Section 4, we develop an abstract semantics which approximates the observable behavior of the program and is adequate for modular data-flow analysis.

We will focus our attention now on a special class of abstract interpretations which are obtained from what we call a *sequence abstraction* $\tau : (\mathcal{C}^*, \leq) \to (AS, \lesssim)$. We require that $(AS, \lesssim)$ is noetherian and that τ is surjective and monotone.

We start by choosing as abstract domain $\mathbb{A} := \mathcal{P}(AS)$, ordered by an extension to sets $X \lesssim_S Y$ iff $\forall x \in X\, \exists y \in Y : (x \lesssim y)$. We will call elements of $\mathbb{A}$ abstract sequences. The concrete domain $\mathbb{E}$ is $\mathcal{P}(\mathcal{C}^*)$, ordered by set inclusion.

Then we can lift τ to a Galois Insertion of $\mathbb{A}$ into $\mathbb{E}$ by defining

$$\alpha(E)(p) := \{\tau(s) \mid s \in E(p)\}$$
$$\gamma(A)(p) := \{s \mid \tau(s) \in A(p)\}$$

for some $\mathit{ProcHead}\ p$. Then, we can lift in the standard way to abstract Interpretations the approximation induced by the above abstraction. The Interpretation $f : \mathit{ProcHeads} \to \mathcal{P}(\mathcal{C}^*)$ can be approximated by $f^\alpha : \mathit{ProcHeads} \to \alpha(\mathcal{P}(\mathcal{C}^*))$.

The only requirement we put on τ is that $\alpha(\mathit{Sem}(\mathcal{D}))$ is finite, where $\mathit{Sem}(\mathcal{D})$ is the (concrete) semantics of $\mathcal{D}$.

Now we can derive the optimal abstract version of $T_{\mathcal{D}}$ as $T^\alpha_{\mathcal{D}} := \alpha \circ T_{\mathcal{D}} \circ \gamma$. By applying the previous definition of α and γ this turns out to be equivalent to the following definition.

Definition 5. *Let τ be a sequence abstraction, $X \in \mathbb{A}$ be an abstract Interpretation. Then,*

$$T^\alpha_{\mathcal{D}}(X)(p(x)) = \tau([\![\exists_y (A \parallel d_{xy})]\!]_X) \text{ if } p(y) :- A \in \mathcal{D}$$

where the abstract denotational semantics is defined in Table 3.

Abstract interpretation theory assures that $T^\alpha_{\mathcal{D}} \uparrow \omega$ is the best correct approximation of $\mathit{Sem}(\mathcal{D})$. Correct means $\alpha(\mathit{Sem}(\mathcal{D})) \sqsubseteq T^\alpha_{\mathcal{D}} \uparrow \omega$ and best means that it is the minimum w.r.t. $\sqsubseteq$ of all correct approximations.

Now we can define the abstract semantics as the least fixpoint of this (continuous) operator.

Definition 6. *The abstract least fixpoint semantics of a program $\mathcal{D}$ is defined as $\mathcal{F}^\alpha(\mathcal{D}) = T^\alpha_{\mathcal{D}} \uparrow \omega$.*

By our finiteness assumption on τ we are guaranteed to reach the fixpoint in a finite number of steps, that is, there exists $h \in \mathbb{N}$ s.t. $T^\alpha_{\mathcal{D}} \uparrow \omega = T^\alpha_{\mathcal{D}} \uparrow h$.

D1	$[\![\mathbf{tell}(c)]\!]^\tau_I \;=\; \tau(\{d.\beta \mid d \models c, \beta \in \mathcal{C}^*\})$
D2	$[\![\sum_{j\in J} \mathbf{when}\ c_j\ \mathbf{do}\ P_j]\!]^\tau_J \;=\; \bigcup_{i\in J} \tau(\{d.\beta \mid d \models c_i, d.\beta \in [\![P_i]\!]^\tau_I\}) \cup \bigcap_{i\in J} \tau(\{d.\beta \mid d \not\models c_i, d.\beta \in \mathcal{C}^*\})$
D3	$[\![P \parallel Q]\!]^\tau_I \;=\; [\![P]\!]^\tau_I \cap [\![Q]\!]^\tau_I$
D4	$[\![\mathbf{local}\ x\ \mathbf{in}\ P]\!]^\tau_I \;=\; \{\beta \mid$ there exists $\beta' \in [\![P]\!]^\tau_I$ s.t. $\exists_x\beta = \exists_x\beta'\}$
D5	$[\![\mathbf{next}\ P]\!]_I \;=\; \tau(\mathcal{C}) \cup \tau(\{d.\beta \mid d \in \mathcal{C}, \beta \in [\![P]\!]_I\})$
D6	$[\![p(x)]\!]^\tau_I \;=\; \tau(I(p(x)))$
D7	$[\![\star P]\!]^\tau_I \;=\; \tau(\{\beta.\beta' \mid \beta \in \mathcal{C}^*, \beta' \in [\![P]\!]^\tau_I\})$

Table 3. Abstract denotational semantics for `ntcc`

5.1 A case study: The domain *sequence*(k)

Now we show how to approximate a set of computed sequences by means of a *sequence*(k) cut [20], i.e., by using a sequence abstraction which approximates sequences having a length bigger than k. Sequences are approximated by replacing each constraint at length bigger than k with the constraint `false`.

First of all we define the sequence abstraction $s/_k$ (for $k \geq 0$) as the *sequence*(k) cut of the concrete sequence s. We mean by that $s/_k = s'$, where $s'(i) = s(i)$ for $i \leq k-1$ and $s'(k) =$ `false`. We denote by $S/_k$ the set of sequences cut at length k. The abstract domain $\mathbb{A}$ is thus $\mathcal{P}(S/_k)$ ordered by the extension of ordering $\leq$ to sets, i.e. $X \;\leq_S\; Y$ iff $\forall x \in X\, \exists y \in Y : (x \;\leq\; y)$ The resulting abstraction α is $\kappa(E)(p) := \{s/_k \mid s \in I(p)\}$. By abuse, we will denote by the same notation $\leq_S$ its standard extension to interpretations. A sequence $s = c_1, \ldots, c_n$ is complete if $c_n \neq false$.

We provide a simple and effective mechanism to compute the abstract fixpoint semantics.

Definition 7. *The effective abstract least fixpoint semantics of a program $\mathcal{D}$, is defined as $\mathcal{F}^\kappa(\mathcal{D}) = T^\kappa_\mathcal{D} \uparrow \omega$.*

Proposition 1 (Correctness). *Let $\mathcal{D}$ be a program and $k > 0$.*

1. $\mathcal{F}^\kappa(\mathcal{D}) \leq_S \kappa(\mathcal{F}(\mathcal{D})) \leq_S \mathcal{F}(\mathcal{D})$.
2. *Let Y be $Fix^\kappa\ \mathcal{D}$, and let Z be $Fix\ \mathcal{D}$.*
   For all $s \in \mathcal{F}^\kappa(\mathcal{D})(Y)(p(x))$ that are complete, $s \in \mathcal{F}(\mathcal{D})(Z)(p(x))$.

6 Abstract diagnosis of `ntcc` programs

In this section, we recall the framework developed in [1]. We modify it in order to be able to handle the specific features of `ntcc`. Then we show that the main correctness and completeness results can be extended to `ntcc`. We also develop a completely new methodology based on the linear temporal logic (LTL) associated to the calculus [15]. The basic idea is that given a specification in the logic we can

generate automatically the sequences which we use for debugging the program. Let us recall the syntax and semantics of this logic (see [15] for further details).

Formulae in the `ntcc` LTL are given by the following grammar:

$$A, B, ... : c \mid A \dot{\Rightarrow} A \mid \dot{\neg} A \mid \dot{\exists}_x A \mid \circ A \mid \Diamond A \mid \Box A$$

c is a constraint. $\dot{\Rightarrow}$, $\dot{\neg}$ and $\dot{\exists}_x$ represent the linear-temporal logic implication, negation and existential quantification, respectively. These symbols should not be confused with their counterpart in the constraint system (i.e $\Rightarrow$, $\neg$ and $\exists$). Symbols $\circ$, $\Box$ and $\Diamond$ denote the temporal operators *next*, *always* and *eventually*.

The interpretation structures of formulae in this logic are infinite sequences of constraints. We say that $\beta \in C^*$ is a model of (or that it satisfies) A, notation $\beta \models A$, if $\langle \beta, 1 \rangle \models A$ where:

$$\begin{array}{ll}
\langle \beta, i \rangle \models c & \text{iff } \beta(i) \models c \\
\langle \beta, i \rangle \models \dot{\neg} A & \text{iff } \langle \beta, i \rangle \not\models A \\
\langle \beta, i \rangle \models A_1 \dot{\Rightarrow} A_2 & \text{iff } \langle \beta, i \rangle \models A_1 \textit{ implies} \langle \beta, i \rangle \models A_2 \\
\langle \beta, i \rangle \models \circ A & \text{iff } \langle \beta, i+1 \rangle \models A \\
\langle \beta, i \rangle \models \Box A & \text{iff } \forall_{j \geq i} \langle \beta, j \rangle \models A \\
\langle \beta, i \rangle \models \Diamond A & \text{iff } \exists_{j \geq i} \textit{ s.t.} \langle \beta, j \rangle \models A \\
\langle \beta, i \rangle \models \dot{\exists}_x A & \text{iff exists } \beta' \text{ s.t. } \exists_x \beta = \exists_x \beta'
\end{array}$$

The LTL is then used to specify (temporal) properties of programs:

Definition 8. *We say that P satisfies some property A written $P \vdash A$, iff $sp(P) \subseteq [\![A]\!]$ where $[\![A]\!]$ is the collection of all models of A, i.e., $[\![A]\!] = \{\beta \models A\}$*

Program properties which can be of interest are Galois Insertions between the concrete domain (the set of interpretations ordered pointwise) and the abstract domain chosen to model the property. The following Definition extends to abstract diagnosis the definitions given in [19,10,13] for declarative diagnosis. In the following, $\mathcal{I}^\alpha$ is the specification of the intended behavior of a program.

Definition 9. *[1] Let $\mathcal{D}$ be a program and α be a property.*

1. *$\mathcal{D}$ is* partially correct *w.r.t. $\mathcal{I}^\alpha$ if $\mathcal{I}^\alpha \sqsubseteq \alpha(Sem(\mathcal{D}))$.*
2. *$\mathcal{D}$ is* complete *w.r.t. $\mathcal{I}^\alpha$ if $\alpha(Sem(\mathcal{D})) \sqsubseteq \mathcal{I}^\alpha$.*
3. *$\mathcal{D}$ is* totally correct *w.r.t. $\mathcal{I}^\alpha$, if it is partially correct and complete.*

Note that the above definition is given in terms of the abstraction of the concrete semantics $\alpha(Sem(\mathcal{D}))$ and not in terms of the (possibly less precise) abstract semantics $Sem^\alpha(\mathcal{D})$. This means that $\mathcal{I}^\alpha$ is the abstraction of the intended concrete semantics of $\mathcal{D}$. In other words, the specifier can only reason in terms of the properties of the expected concrete semantics without being concerned with (approximate) abstract computations.

The *diagnosis* determines the "basic" symptoms and, in the case of incorrectness, the relevant rule in the program. This is modeled by the definitions of *abstractly incorrect rule* and *abstract uncovered equation*.

Definition 10. *Let r be a procedure definition. Then r is* abstractly incorrect *if* $\exists p(x) : -A \in \mathcal{D}.T^{\alpha}_{\{r\}}(\mathcal{I}^{\alpha})(p(x)) \not\sqsubseteq \mathcal{I}^{\alpha}$.

Informally, r is abstractly incorrect if it derives a wrong abstract element from the intended semantics.

Definition 11. *Let $\mathcal{D}$ be a program. $\mathcal{D}$ has abstract uncovered elements if* $\exists p(x) : -A \in \mathcal{D}.\mathcal{I}^{\alpha} \not\sqsubseteq T^{\alpha}_{\mathcal{D}}(\mathcal{I}^{\alpha})(p(x))$.

Informally, a sequence s is uncovered if there are no rules deriving it from the intended semantics. It is worth noting that checking the conditions of Definitions 10 and 11 requires one application of $T^{\alpha}_{\mathcal{D}}$ to $\mathcal{I}^{\alpha}$, while the standard detection based on symptoms [19] would require the construction of $\alpha(Sem(\mathcal{D}))$ and therefore a fixpoint computation.

Now, we want to recall the properties of the diagnosis method. The proofs of the following theorems in this section are extensions of those in [1].

Theorem 1. *[1] If there are no abstractly incorrect rules in $\mathcal{D}$, then $\mathcal{D}$ is partially correct w.r.t. $\mathcal{I}^{\alpha}$.*

Theorem 2. *[1] Let $\mathcal{D}$ be partially correct w.r.t. $\mathcal{I}^{\alpha}$. If $\mathcal{D}$ has abstract uncovered elements then $\mathcal{D}$ is not complete.*

Abstract incorrect rules are in general just a hint about a possible source of errors. If an abstract incorrect rule is detected, one would still have to check on the abstraction of the concrete semantics if there is indeed a bug. This is obviously unfeasible in an automatic way. However we will see that, by adding to the scheme an under-approximation of the intended specification, something worthwhile can still be done.

Real errors can be expressed as incorrect rules according to the following definition.

Definition 12. *Let r be a process definition. Then r is* incorrect *if there exists a sequence s such that $s \in T_{\{r\}}(\mathcal{I})$ and $s \notin \mathcal{I}$.*

Definition 13. *Let $\mathcal{D}$ be a program. Then $\mathcal{D}$ has an* uncovered element *if there exists a sequence s such that $s \in \mathcal{I}$ and $s \notin T_{\mathcal{D}}(\mathcal{I})$.*

The check of Definition 12 (as claimed above) is not effective. This task can be (partially) accomplished by an automatic tool by choosing a suitable under-approximation $\mathcal{I}^c$ of the specification $\mathcal{I}$, $\gamma(\mathcal{I}^c) \subseteq \mathcal{I}$ (hence $\alpha(\mathcal{I}) \sqsubseteq \mathcal{I}^c$), and checking the behavior of an abstractly incorrect rule against it.

Definition 14. *Let r be a process definition. Then r is* provably incorrect *using α if* $\exists p(x) : -A \in \mathcal{D}.T^{\alpha}_{\{r\}}(\alpha(\mathcal{I}^c))(p(x)) \not\sqsubseteq \mathcal{I}^{\alpha}$.

Definition 15. *Let $\mathcal{D}$ be a program. Then $\mathcal{D}$ has* provably uncovered *elements using α if* $\exists p(x) : -A \in \mathcal{D}.\alpha(\mathcal{I}^c) \not\sqsubseteq T^{\alpha}_{\mathcal{D}}(\mathcal{I}^{\alpha})(p(x))$.

The name "provably incorrect using α" is justified by the following theorem.

Theorem 3. *[1] Let r be a program rule (process) and $\mathcal{I}^c$ such that $(\gamma\alpha)(\mathcal{I}^c) = \mathcal{I}^c$. Then if r is provably incorrect using α it is also incorrect.*

By choosing a suitable under-approximation we can refine the check for wrong rules. For all abstractly incorrect rules we check if they are provably incorrect using α. If it so then we report an error, otherwise we can just issue a warning.

As we will see in the following, this property holds (for example) for our case study. By Proposition 1 the condition $(\gamma\alpha)(\mathcal{I}^c) = \mathcal{I}^c$ is trivially satisfied by any subset of the 'finite' sequences in the over-approximation. We can also consider a finite subset of the sequences in which we do the following change: if a sequence is such that all its elements starting from the length k are equal to **false**, we replace such elements by **true**.

Theorem 4. *[1] Let $\mathcal{D}$ be a program. If $\mathcal{D}$ has a provably uncovered element using α, then $\mathcal{D}$ is not complete.*

Abstract uncovered elements are provably uncovered using α. However, Theorem 4 allows us to catch other incompleteness bugs that cannot be detected by using Theorem 2 since there are provably uncovered elements using α which are not abstractly uncovered.

The diagnosis w.r.t. approximate properties is always effective, because the abstract specification is finite. As one can expect, the results may be weaker than those that can be achieved on concrete domains just because of approximation:

- every incorrectness error is identified by an abstractly incorrect rule. However an abstractly incorrect rule does not always correspond to a bug. Anyway,
- every abstractly incorrect rule which is provably incorrect using α corresponds to an error.
- provably uncovered equations always correspond to incompleteness bugs.
- there exists no sufficient condition for completeness.

6.1 Our case study

An efficient debugger can be based on the notion of over-approximation and under-approximation for the intended fixpoint semantics that we have introduced. The basic idea is to consider two sets to verify partial correctness and determine program bugs: $\mathcal{I}^\alpha$ which over-approximates the intended semantics $\mathcal{I}$ (that is, $\mathcal{I} \subseteq \gamma(\mathcal{I}^\alpha)$) and $\mathcal{I}^c$ which under-approximates $\mathcal{I}$ (that is, $\gamma(\mathcal{I}^c) \subseteq \mathcal{I}$).

Let us now derive an efficient debugger by choosing suitable instances of our general framework. Let α be the *sequence*(k) abstraction κ of the program that we have defined in previous section. Thus we choose $\mathcal{I}^\kappa = \mathcal{F}^\kappa(\mathcal{I})$ as an over-approximation of the values of a program. We can consider any of the sets defined in the works of [4,7] as an under-approximation of $\mathcal{I}$. In concrete, we take the finite abstract sequences of $\mathcal{I}^\kappa$ as $\mathcal{I}^c$ but replacing the constraint **false** by **true** after the position κ. This provides a simple albeit useful debugging scheme which is satisfactory in practice.

Another possibility is to generate the sequences for the under and the over-approximation from the sequences that are models for a LTL formula specifying the intended behavior of the program. Let us illustrate the method by using the guiding example.

Example 2. Let first consider what the intended behaviour of the system proposed in Example 1 should be. We require that as soon as the constraint $failure$ can be entailed from the store, the system must exhibit the constraint *stop*. This specification can be provided by means of another (correct) program or by a formula in the LTL. Let us explore both cases.

Let $control'$ and $action'$ be the intended system:

$$control' = \textbf{when } failure \textbf{ do } action \mid\mid \textbf{next } control'$$
$$action' = \textbf{tell}(stop)$$

and A be the LTL formula:

$$A = \Box(failure \Rightarrow stop)$$

Sequences generated by A and by the processes $control'$ and $action'$ coincides and can be used to compute the intended behavior $\mathcal{I}$, the corresponding over-approximation $\mathcal{I}^\alpha = \mathcal{I}^\kappa$ and the under-approximation $\mathcal{I}^c$ as was defined previously. The intended behaviour $\mathcal{I}$ is then:

$\mathcal{I}(control') = \{s'.d.s''|d \models failure \Rightarrow d.s'' \in [\![action']\!]_{\mathcal{I}}\}$ for any s' and s''
$\mathcal{I}(action') = \{d.s|d \models stop\}$

According to Definition 14, let us compute $T^\alpha_{\{control\}}(\alpha(\mathcal{I}^c))(control)$:

$$T^\alpha_{\{control\}}(\alpha(\mathcal{I}^c))(control) = \tau(\{d_1...d_k.\textbf{true}^*|d_1 \models failure \Rightarrow d_2 \models stop \textit{ and } \forall_{i>1} d_i \models failure \Rightarrow d_i \models stop\})$$

Given $s_1 = failure.stop.\texttt{true}^* \in T^\alpha_{\{control\}}(\alpha(\mathcal{I}^c))(control)$ and $s_2 = failure \wedge stop.\texttt{true}^* \in \mathcal{I}^\alpha$, notice that $s_1 \not\leq s_2$ suggesting that *control* is provably incorrect. The error is due to it delays one time unit the call to the process *action*.

Example 3. In this example we debug a program capturing the behaviour of a RCX-based robot that can go right or left. The movement of the robot is defined by the following rules: 1) If the robot goes right in the current time unit, it must turn left in the next one. 2) After two consecutive time units moving to the left, next decision must be turn right. The program proposed is depicted below:

$$\begin{aligned}
\textbf{GoR} &= \textbf{tell}(a_0 = r) || \textbf{next tell}(a_1 = r) \\
\textbf{GoL} &= \textbf{tell}(a_0 = l) || \textbf{next tell}(a_1 = l) \\
\textbf{Update} &= \textbf{when } a_1 = l \textbf{ do next tell}(a_2 = l) + \\
&\quad \textbf{when } a_1 = r \textbf{ do next tell}(a_2 = r) \\
\textbf{Zigzag} &= \textbf{when } a_2 \neq r \textbf{ do GoR} + \textbf{when } a_2 \neq l \textbf{ do GoL} || \\
&\quad \textbf{Update} || \textbf{next Zigzag}
\end{aligned}$$

GoR and **GoL** represent the decision of turning right or left respectively. **Update** keeps track of the second-to-last action and **ZigZag** controls the behaviour of the robot. Variables a_0, a_1 and a_2 represent the current, the previous and the second-to-last decisions respectively.

Now we provide the intended specification for each process:

$$\begin{array}{ll} \mathcal{I}(GoR) & = \{s|\forall_i.\ s(i) \models (a_0 = r) \text{ and } s(i+1) \models (a_1 = r)\} \\ \mathcal{I}(GoL) & = \{s|\forall_i.\ s(i) \models (a_0 = l) \text{ and } s(i+1) \models (a_1 = l)\} \\ \mathcal{I}(Update) & = \{s|\forall_i.\ s(i) \models (a_1 = l) \Rightarrow s(i+1) \models (a_2 = l)\} \cup \\ & \quad \{s|\forall_i.\ s(i) \models (a_1 = r) \Rightarrow s(i+1) \models (a_2 = r)\} \end{array}$$

Specification of **Zigzag** can be stated by the following LTL formula:

$$A = \Box((a_0 = r) \dot{\Rightarrow} \circ(a_0 = l) \dot{\wedge} ((a_0 = l) \dot{\wedge} \circ(a_0 = l) \dot{\Rightarrow} \circ \circ (a_0 = r)))$$

The property states that always is true that 1)if the current decision is go right, the next decision must be go left and 2)if the robot goes left in two consecutive time units, the next decision must be turn right. $\mathcal{I}(Zigzag)$ can be then viewed as all the possible models of the formula A, i.e. $[\![A]\!]$:

$$\begin{array}{ll} \mathcal{I}(Zigzag) = & \{s|s(i) \models (a_0 = r) \Rightarrow s(i+1) \models (a_0 = l)\} \cup \\ & \{s|\ (s(i) \models (a_0 = l) \wedge s(i+1) \models (a_0 = l)) \Rightarrow s(i+2) \models (a_0 = r)\} \end{array}$$

Let $\mathcal{I}^\alpha$ be the abstraction $\mathcal{I}^\kappa$ representing the over-approximation of $\mathcal{I}$ and $\mathcal{I}^c$ the under-approximation as before. Let us compute a step of $T^\alpha_{\{Zigzag\}}$:

$$\begin{array}{ll} T^\alpha_{\{Zigzag\}}(\alpha(\mathcal{I}^c))(Zigzag) = & [\![\textbf{when } a_2 \neq r \textbf{ do GoR} + \textbf{when } a_2 \neq l \textbf{ do GoL}|| \\ & \quad \textbf{Update}]\!]_{\alpha(\mathcal{I}^c)} \\ & = (\{d_1.s|d_1 \models (a_2 \neq r) \Rightarrow d_1.s \in [\![GoR]\!]_{\alpha(\mathcal{I}^c)}\} \cup \\ & \quad \{d_1.s|d_1 \models (a_2 \neq l) \Rightarrow d_1.s \in [\![GoL]\!]_{\alpha(\mathcal{I}^c)}\}) \cap \\ & \quad \{d.s|s \in [\![Zigzag]\!]_{\alpha(\mathcal{I}^c)}\} \cap [\![\textbf{Update}]\!]_{\alpha(\mathcal{I}^c)} \end{array}$$

Notice that the sequence $d_1.d_2$ where $d_1 \models (a_0 = r)$ and $d_2 \models (a_0 = r)$ is a sequence that can be generated from $T^\alpha_{\{Zigzag\}}(\alpha(\mathcal{I}^c))(Zigzag)$ but $d_1.d_2 \notin \mathcal{I}^\alpha$. Using Definition 14 we can say that $Zigzag$ is provably incorrect. The bug in this case, is that guard in the sub-process **when** $a_2 \neq r$ **do GoR** must be $a_1 \neq r$.

7 Conclusions

We have presented a framework for the declarative debugging of `tcc` programs w.r.t. the set of computed constraint sequences. Our method is based on a denotational semantics for `tcc` programs which models the semantics in a compositional manner. Moreover, we have shown that it is possible to use a linear temporal logic for providing a specification of correct programs, and generate automatically from it the sequences which are necessary for debugging.

We follow the idea of considering declarative specifications as programs. The intended specification can then be automatically abstracted and can be used to automatically debug the final program. As future work we are developing a prototype of our debugging system on top of an implementation of an interpreter of the `ntcc`calculus that we developed in [16].

References

1. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In M. Leuschel and F. Bueno, editors, *Proc. of LOPSTR 2002*, pages 1–16. Springer LNCS 2664, 2003.
2. M. Alpuente, F. Correa, and M. Falaschi. A Debugging Scheme for Functional Logic Programs. In M. Hanus, editor, *Proc. of WFLP 2001*, volume 64 of *ENTCS*. Elsevier, 2002.
3. S. Antoy and S. Johnson. TeaBag: A Functional Logic Debugger. In H. Kuchen, editor, *Proc. of WFLP 2004*, pages 4–18, 2004.
4. F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M Hermenegildo, J. Maluszyński, and G. Puebla. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In *Proc. of AADEBUG'97*, pages 155–170. U. of Linkoping Press, 1997.
5. M. Cameron, M. García de la Banda, K. Marriott, and P. Moulder. Vimer: a visual debugger for mercury. In *Proc. of the 5th PPDP*, pages 56–66. ACM Press, 2003.
6. M. Comini, R. Gori, and G. Levi. Assertion based Inductive Verification Methods for Logic Programs. In A. K. Seda, editor, *Proceedings of MFCSIT'2000*, volume 40 of *ENTCS*. Elsevier Science Publishers, 2001.
7. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of Logic Programs by Abstract Diagnosis. In M. Dams, editor, *Proc. of LOMAPS'96*, volume 1192 of *LNCS*, pages 22–50, Berlin, 1996. Springer-Verlag.
8. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.
9. F. S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM TOPLAS*, 19(5):685–725, 1997.
10. G. Ferrand. Error Diagnosis in Logic Programming, an Adaptation of E.Y.Shapiro's Method. *Journal of Logic Programming*, 4(3):177–198, 1987.
11. M. Hanus and B. Josephs. A debugging model for functional logic programs. *Proc. of 5th PLILP*, volume 714 of *LNCS*, pages 28–43. Springer, 1993.
12. M. Hanus and J. Koj. An integrated development environment for declarative multi-paradigm programming. In *Proc. of the International Workshop on Logic Programming Environments (WLPE'01)*, pages 1–14, Paphos (Cyprus), 2001.
13. J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.
14. L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18(1):401–408, 1996.
15. M. Nielsen, C. Palamidessi, and F.D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing*, 9(1):145–188, 2002.
16. C. Olarte, C. Rueda. A Stochastic Concurrent Constraint Based Framework to Model and Verify Biological Systems. Clei Electronic Journal Vol 9:2, 2005.
17. V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80. IEEE Computer Society Press, 1994.
18. V.A. Saraswat. Semantic Foundation of Concurrent Constraint Programming. In *Proc. of 18th ACM POPL*. ACM, New York, 1991.
19. E. Y. Shaphiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Massachusetts, 1982. ACM Distinguished Dissertation.
20. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In S. Tärnlund, editor, *Proc. of 2nd ICLP*, pages 127–139, 1984.