Set-based Error Diagnosis of Concurrent Constraint Programs

Andreas Podelski, Witold Charatonik
Max-Planck-Institut für Informatik
D-66123 Saarbrücken, Germany
{podelski ; witold}@mpi-sb.mpg.de
Martin Müller
Programming Systems Lab, Universität des Saarlandes
D-66041 Saarbrücken, Germany
mmueller@ps.uni-sb.de

Abstract

We present an automated method for the static prediction of the runtime error 'deadlock or failure' in concurrent constraint programs. Operationally, the method is based on a new set-based analysis of reactive logic programs which computes an approximation of the greatest-model semantics. Semantically, the method is based on the connection between the inevitability of 'deadlock or failure' in concurrent constraint programs, finite failure in logic programming and the greatest-model semantics over infinite trees.

1 Introduction

The concurrent constraint (cc) paradigm is a powerful concurrent programming paradigm in which the – superficially unrelated – concepts of concurrency and constraints fruitfully interact (for entry points to the vast literature, see [25, 26, 8, 7]). On the one hand, concurrency can be a useful programming abstraction for the hard task of writing constraint solvers. On the other hand, the intrinsicly complex synchronization of concurrent processes can be transferred to high-level, conceptually simpler logical constraint problems. The cc paradigm makes it possible to turn these two reciprocal dependencies into a useful programming tool. By now, several implementations exist; we are interested in particular in contributing to the development of the freely available Oz System [19, 26].

It is evident that any kind of static analysis is desirable that can detect errors or increase confidence in the correctness of programs. This holds in general, and even more so in the traditionally untyped setting of concurrent constraint programming. More generally, we believe that concurrent constraint languages have a high potential for automated verification because of the close connection with logic: computation states are described by constraints and concurrent composition corresponds to conjunction.

In this paper, we first present a new set-based analysis for logic programs

that are intended to run possibly forever. We call such programs *reactive*. We then apply the set-based analysis to the automated and static prediction of an important kind of errors in cc programs, namely deadlock or failure.

Logic programs. The denotational semantics of a reactive logic program P is defined by the greatest fixpoint of the T_P operator, which also is the greatest model of Clark's completion of P [18, 4], over the domain of infinite trees. Reactive logic programs are called "perpetual processes" in [18]; we use the term "reactive" coined in [24] referring to possibly non-terminating behavior. The set-based analysis that we introduce uses *co-definite set constraints* [2]. In a first step, the analysis derives a co-definite set constraint φ_P whose greatest solution approximates the greatest model of P. In a second step, the analysis applies the algorithm for solving co-definite set constraints from [2] and computes a representation of the greatest solution of φ_P. The values under this solution are sets of infinite trees; intuitively, these sets describe supersets of the sets of possible runtime values in possibly non-terminating executions of the program P. The values are, however, also relevant for predicting a certain runtime behavior of P, as we explain next.

One can characterize finite failure of (possibly non-ground) derivations over infinite trees precisely through the greatest model (see Theorem 3.1). This characterization allows us to use the computed greatest solution of the derived co-definite set constraint φ_P for diagnosing finite failure. Namely, every fair derivation of the query $p(x)$ finitely fails if the value of the corresponding variable is the empty set. (Note that finite failure over infinite trees entails finite failure over finite trees.)

Concurrent constraint (cc) programs. We may carry over the approximation result of the greatest model to cc programs. Its interpretation in operational terms, however, needs some more care due to the possibility of deadlock. The characterization of finite failure through the greatest model still holds for cc programs without deadlock (Theorem 5.4). In general we cannot statically exclude deadlock.

In cc programs, an inconsistent constraint store (*viz.*, failure) is a run-time error. This is in contrast with logic programming where failure is part of the backtracking mechanism. Deadlock is a second kind of runtime error. If we do not distinguish between either kind of error, i.e., take the disjunction 'deadlock or failure', then we can apply the set-based diagnosis of finite failure for logic programs. In summary, emptiness under the greatest solution of the co-definite set constraint derived from the cc program P is a sufficient condition for the inevitability of the run-time error 'deadlock or failure'. Or, more precisely, its computed greatest solution describes, for each input variable x, a superset of the possible values of x in non-erroneous input states.

The following sections investigate, respectively, some example programs, finite failure for logic programs, the error 'deadlock or failure' in cc programs, the new set-based analysis together with its application as a diagnosis method, related and, in conclusion, future work.

2 Examples

The following examples illustrate how our method of approximating greatest models with co-definite set constraints tests the inevitability of certain runtime errors. We first take the very simple program

$$\begin{array}{l} p(X) \text{:-}\ q(X), r(X). \\ q(a). \\ r(b). \end{array}$$

The derived set constraint is, simplified, $p \subseteq q \wedge q \subseteq a \wedge p \subseteq r \wedge r \subseteq b$. Its greatest solution for p is the empty set (and indeed, the query $p(x)$ finitely fails). Similarly, the program

$$\begin{array}{l} p(f(\_, X)) \text{:-}\ q(f(\_, X)). \\ q(f(\_, X)) \text{:-}\ p(g(\_, X)). \end{array}$$

yields the set constraint $p \subseteq f(\_, x) \wedge x \subseteq f^{-1}_{(2)}(q) \wedge q \subseteq f(\_, x) \wedge x \subseteq g^{-1}_{(2)}(p)$, whose greatest solution for p is the empty set (since $g^{-1}_{(2)}(f(M_1, M_2)) = \emptyset$ for any set M_1, M_2). Now consider the following simplified stream program.

$$\begin{array}{l} \mathsf{stream}([X, Y|S]) :-\ Y = s(s(X)),\ \mathsf{computation}(Y),\ \mathsf{stream}([Y|S]). \\ \mathsf{main}(Z) :-\ \mathsf{stream}([Z|T]). \end{array}$$

Suppose we know that the predicate computation makes sense only for (trees representing) odd numbers, whereas no such restriction is known for main and stream. This invariant can be expressed by the following set constraint, which may have been derived from another code fragment or externally provided by a program annotation.

$$\mathsf{computation} \subseteq s(0) \cup s(s(\mathsf{computation})) \tag{1}$$

We can approximate the set of non-failed computations of the program with the constraint (where $s^{-1}_{(1)}(M) = \{t \mid s(t) \in M\}$, and similarly $cons^{-1}_{(1)}(M)$ extracts the heads of all lists in M, and $cons^{-1}_{(2)}(M)$ the tails):

$$\begin{array}{l} \mathsf{stream} \subseteq cons(X, cons(Y, S)) \ \wedge \\ X \subseteq \mathsf{computation}\ \wedge\ X \subseteq s^{-1}_{(1)}(s^{-1}_{(1)}(Y)) \ \wedge \\ Y \subseteq cons^{-1}_{(1)}(\mathsf{stream})\ \wedge\ S \subseteq cons^{-1}_{(2)}(\mathsf{stream}) \ \wedge \\ \mathsf{main} \subseteq cons^{-1}_{(1)}(\mathsf{stream})\ . \end{array} \tag{2}$$

It is not difficult to see that the greatest solution of the conjunction of (1) and (2) assigns to the variable main (as well as to X, Y, and computation) the set of odd numbers. We obtain from this fact that, for example, the query main(0) inevitably leads to a state where computation is called with a wrong argument.

We now illustrate the necessity to consider infinite trees by another example. Consider the reactive logic program P defined by $p(f(x)) :- p(x)$. The execution of the non-ground query $p(x)$ does not fail, over the domain of finite as well as over the domain of infinite trees. We derive the co-definite set constraint $\varphi_P \equiv p \subseteq f(x) \wedge x \subseteq p$. When interpreted over sets of finite trees, the greatest solution of φ_P is the valuation assigning the empty set to p. In the infinite tree case the greatest solution assigns to p the singleton set containing the infinite tree $\{f(f(f(\ldots)))\}$. That is, an interpretation of the derived co-definite set constraint over sets of finite trees does not admit a conclusion about finite failure of non-ground queries.

3 Logic Programs

Notation. We assume a ranked alphabet Σ fixing the arity $n \geq 0$ of its function symbols $f, g, \ldots$ and constant symbols $a, b, \ldots$, and an infinite set Var of variables $x, y, z, \ldots$. We write $\bar{x}$ for finite sequences of variables (whose length equals the arity of f in $f(\bar{x})$. The set of terms over Σ and Var is $T_\Sigma(\mathsf{Var})$;a term without variables is called a *ground term*. The set of *infinite trees* over Σ is T_Σ^∞. By convention, an infinite tree is a tree whose branches are finite or infinite. We use the meta-variable t to refer to both, trees and terms. We write as $\exists_{-x}\Phi$ for the existential closure of the formula Φ with respect to all variables in Φ but x. We also assume a set Pred of predicate symbols p. For better readability, we assume that all predicates are unary; the results can easily be extended to the case without this restriction (for example, by adding symbols that form tuples).

The *Herbrand Base* $\mathcal{B}$ is the set of all ground atoms over Pred and T_Σ^∞, *i.e.*, $\mathcal{B} = \{p(t) \mid p \in \mathsf{Pred}, t \in T_\Sigma^\infty\}$.[1]

A *logic program* defines predicates through *clauses* of the form

$$p(t) :- p_1(t_1), \ldots, p_n(t_n)\,.$$

A complete program P is a set consisting of n_p clauses for each predicate $p \in \mathsf{Pred}$, each with the head $p(t_i)$ and the body consisting of $n_{i,p}$ atoms $p_{ij}(t_{ij})$; we leave the index p as in t_{ijp} implicit for better readability.

We always implicitly refer to the *completion of* P [4] for the logical semantics of P.

$$P \quad \equiv \quad \bigwedge_{p \in \mathsf{Pred}} \forall x \;\; p(x) \leftrightarrow \bigvee_{i=1}^{n_p} \exists_{-x}(\; x = t_i \wedge \bigwedge_{j=1}^{n_{i,p}} p_{ij}(t_{ij}) \;)$$

A *query* s is a conjunction $\bigwedge_k p_k(t_k)$ where the t_k are terms. A *ground query* is a query $\bigwedge_k p_k(t_k)$ such that all t_k are ground. We use the predicate

[1] What we call Herbrand Base is sometimes called *Complete Herbrand Base* [18], namely when it needs to be distinguished from the set of ground atoms over Pred and *finite* trees.

constant *true* as the neutral element for conjunction: *i.e.*, $s = s \wedge true$. In particular, the "empty query" is written as *true*.

An *interpretation* ρ (sometimes called a *model*) is a subset of the Herbrand Base, $\rho \subseteq \mathcal{B}$. Interpretations are ordered by subset inclusion. We identify an interpretation ρ with the valuation $\rho : \mathsf{Pred} \to 2^{T_\Sigma^\infty}$ from predicate symbols to sets of trees where $\rho(p) = \{t \in T_\Sigma^\infty \mid p(t) \in \rho\}$. A *model of the program* P is a valuation such that the formula P is valid (in the usual sense of logic). The greatest model of P, denoted by $gm(P)$, always exists.

Operational Semantics. The logic program P defines a *fair transition system* $\mathcal{T}_P = \langle \mathcal{S}, \tau_P \rangle$ whose one-step transition relation is defined as usual for derivations of logic programs or constraint logic programs [18, 16], with the fair non-deterministic selection rule. Conjunction is operationally parallel composition, and disjunction is non-deterministic choice. The non-determinism of the selection rule translates to the non-determinism of the interleaving semantics; the fairness of the selection rule is exactly the fairness of the transition system (which is thus captured precisely by the greatest-model semantics). The set $\mathcal{S}$ of states consists of the possibly non-ground queries (including *true*), and the *failure* state *false*:[2]

$$\mathcal{S} \quad = \quad \{\bigwedge_k p_k(t_k) \mid \forall k \;\; p_k \in \mathsf{Pred},\; t_k \in T_\Sigma(\mathsf{Var})\} \cup \{false\}$$

Similarly, P defines a *fair ground transition system* $\mathcal{T}_P^g = \langle \mathcal{S}^g, \tau_P^g \rangle$. The states in S^g are the ground queries including *true* and *false*. The transition relation τ_P^g modifies the one of $\mathcal{T}_P$ such that after every step of $\mathcal{T}_P$ all variables in the obtained state are substituted with ground terms.

We say that a derivation *succeeds* [*fails*] if if it ends in the state *true* [*false*]. A query $p(t)$ *succeeds* if there exists a derivation leading to the state *true*; it *finitely fails* if every (fair) $\mathcal{T}_P$ derivation starting with query $p(t)$ leads to failure after finitely many steps. Given the program P, we denote $FF(P)$ the set of all queries $p(t)$ that finitely failed, and $GFF(P)$ the set of all ground queries $p(t) \in \mathcal{B}$ that are *ground finitely failed*; i.e., every T_P^g derivation starting from $p(t)$ reaches failure after finitely many steps. (Note that in general, ground finite failure does not imply finite failure of some ground instance.)

We can characterize the finite failure set of a program P over the domain T_Σ^∞ of infinite trees through the greatest model of P.

[2] The notion of queries considered here is not general enough to model arbitrary execution states of a constraint program over infinite trees. Since terms are finite, we can use them only to model non-cyclic constraint stores. We gloss over this detail here since it is not essential and repairing it would make the notation clumsy (see, however, [21]).

Theorem 3.1 (Characterization of finite failure over infinite trees) Given a logic program P over infinite trees, the query $p(x)$ is finitely failed if and only if the value of p in the greatest model of P over the domain T_Σ^∞ of infinite trees is the empty set; *i.e.*, $p(x) \in FF(P)$ iff $gm(P)(p) = \emptyset$.

Proof. The only-if direction (the algebraic soundness of finite failure [18, 16]) is clear from the fact that $gm(P)(p) = \{t \mid p(t) \notin GFF(P)\}$.

For the other direction, first note that equations over infinite trees have the *saturation property*, which is: an infinite set of constraints is satisfiable if every of its finite subsets is [18, 17, 23].

Now assume that $p(x) \notin FF(P)$. It is sufficient to show that there exists an infinite tree t such that $p(t) \notin GFF(P)$. There exists an execution starting in the state $s = p(x)$ that does not lead to the failure state. That is, there exists a transition sequence $s = s_0, s_1, s_2, \ldots$ such that (in the terminology of constraint logic programming [16]) the constraint store φ_i of every state s_i is satisfiable. Since φ_i is stronger than φ_{i-1} for $i \geq 1$, this implies that $\bigwedge_{i=0}^{n} \varphi_i$ is satisfiable for all n. The saturation property yields that also $\{\varphi_i \mid i \geq 0\}$ is satisfiable. Let α be a solution of $\{\varphi_i \mid i \geq 0\}$. The transition sequence $s'_o, s'_1, s'_2, \ldots$ that we obtain by instantiating the states s_i by the valuation α is a ground transition that does not lead to the fail state. Hence, if $\alpha(x) = t$, then $p(t) \notin GFF(P)$. □

Discussion. It would be surprising if the above theorem was a new observation; since we have not found it in the literature, however, we feel obliged to give its proof.

For comparison, Palmgren [23] has shown that every constraint logic program over a constraint domain with the saturation property is canonical. That is, $gfp(T_P) = T_P{\downarrow}^\omega$ holds (where $T_P{\downarrow}^\omega \bigcap_{i=1}^{\omega} T_P^i(\mathcal{B})$). Since $gfp(T_P) = \mathcal{B} \setminus GFF(P)$ holds, this is sufficient to characterize *ground* finite failure over infinite trees (see also [22]). It does not, however, yield the above theorem. For illustration, consider the example (from [15]) $p(f(x)) :- p(x)$. This program is canonical over finite trees. Its greatest model over finite trees assigns p the empty set. Thus $p(t) \in GFF(P)$ for all *finite* tree t, but $p(x) \notin FF(P)$, in violation of the statement in the theorem.

A similar remark applies to Jaffar and Stuckey's result [17] that $T_P \downarrow^\omega$ equals $\mathcal{B} \setminus [FF(P)]$ where $[FF(P)]$ is the set of *ground instances* of elements of $FF(P)$. This holds for all canonical programs over a solution compact domain, and again the above example applies.

Finally, we want to mention that on can also come up with an alternative (though, less direct and less natural) proof of Theorem 3.1 which uses Palmgren's result [23] and the algebraic completeness of finite failure of ground goals wrt. canonical programs. Here, the trick is to add a clause of the form $main :- p(x)$ to the program P.

4 Concurrent Constraint Programs

We consider *concurrent constraint (*cc*) programs* (see, *e.g.*, [25, 26]) in a normalized form such that we can employ a Prolog-style clausal syntax. This is convenient when we establish a direct connection to logic programming. We assume that every procedure p is defined either by a single fact $p(t)$. or by several *guarded clauses* of the form

$$p(x) :- \ x = t \parallel p_1(t_1), \ldots, p_n(t_n).$$

In such a guarded clause, we call $x = t$ the *guard* and $p_1(t_1), \ldots, p_n(t_n)$ the *body*.

Since we are interested here only in the case where constraints C are term equations interpreted over infinite trees, as in the cc programming language and system Oz [19, 26], the syntactic restriction that we enforce is a proper one only in isolated cases (see Footnote 2). Generally, we can replace a *tell* operation (e.g., the equation $t = t'$ in the body of a clause in a cc program in the style of [25]) by the call of procedures defined by facts.

The operational semantics of a cc program P is defined through a fair transition system $\mathcal{T}_P^{\mathsf{cc}}$ which is the same as the one for logic programs except for one difference: a selected atom $p(t')$ in a query can be applied only if there is a clause, say, $p(x) :- \ x = t \parallel p_1(t_1), \ldots, p_n(t_n).$, such that $x = t'$ entails $\exists_{-x} \ x = t_i$. The successor state of the query is, if the entailment holds, defined in the same way as it is defined for a logic program with the clause $p(t) :- \ p_1(t_1), \ldots, p_n(t_n)$. (For more definitions, see, *e.g.*, [25, 26]).

Failure vs. Deadlock. An execution sequence *deadlocks* if it contains a query $p(t)$ that is never applied because none of the corresponding guards is ever entailed. Note the analogy with failure: an execution sequence *fails* if it contains a query $p(t)$ that is never applied because it does not unify with any of the heads of the clauses of p. Either, failure or deadlock, is a run-time error in cc systems. (Our method cannot directly be applied to a program where deadlock is a desired feature of its execution behavior, and not a bug. In our programming experience, however, this is rarely the case.) We want to give a conservative approximation of all (initial) execution states of cc programs for which either failure or deadlock is inevitable (i.e., every possible fair execution sequence finally reaches a state in which, for fairness reasons, an atoms needs to but cannot be applied, either because no guard is entailed or because the successor state would be *false*). We can express the set of all these states formally through a CTL operator, namely

$$\mathsf{AF}(\{false\}) = \{s \in \mathcal{S} \mid \text{every fair execution starting in } s \text{ deadlocks or fails}\}$$

if we use the following convention. In every fair execution containing the state $s = \bigwedge_k p_k(t_k)$, each atom $p_k(t_k)$ will be *selected* after a finite number of steps. When the atom gets selected in, say, the state s', then it must be applied; if no clause is applicable, then the only successor state of s' is *false*.

We obtain the logic program $\tilde{P}$ from the cc program P by replacing each clause $p(x) :- x = t \parallel p_1(t_1), \ldots, p_n(t_n).$ with $p(t) :- p_1(t_1), \ldots, p_n(t_n).$ (which amounts to replacing the guard operator $\parallel$ with conjunction). This is an abstraction in the following sense.

Proposition 4.1 If the query $p(t)$ finitely fails wrt. the logic program $\tilde{P}$ obtained from the cc program P then it either deadlocks or fails wrt. P. If the query $p(t)$ succeeds wrt. P then it also succeeds wrt. $\tilde{P}$.

Proof. Observe that every (finite or infinite) fair computation in P which neither fails nor deadlocks induces a computation in $\tilde{P}$ which does not fail or deadlock, either. This can be made formal by a simulation argument which exploits that whenever a selected query $p(t)$ is applied with a guarded clause in P it can also be applied with the associated unguarded clause in $\tilde{P}$. This proves the second claim immediately, and also the first one by contraposition. □

Proposition 4.2 (Characterization of finite failure or deadlock) Given a concurrent constraint program P over infinite trees, the query $p(x)$ inevitably either deadlocks or fails in every fair execution if and only if the value of p in the greatest model of $\tilde{P}$ over the domain T_Σ^∞ of infinite trees is the empty set.

Proof. We combine Proposition 4.1 and Theorem 3.1. □

5 Set-based Analysis

Before we introduce the new set-based analysis, we need to recall the definition of co-definite set constraints from [2].

A *set expression* e is built up from first-order terms, union, intersection, complement, and the projection operator.

$$e \quad ::= \quad x \mid f(\overline{e}) \mid e \cup e' \mid e \cap e' \mid e^c \mid f^{-1}_{(k)}(e)$$

If e does not contain the complement operator, then e is called a *positive* set expression. A *set constraint* (in the sense of [12]) is a conjunction of inclusions of the form $e \subseteq e'$.

Definition 5.1 A *co-definite* set constraint φ is a conjunction of inclusions $e_l \subseteq e_r$ between positive set expressions, where the set expressions e_l on the left-hand side of $\subseteq$ are further restricted to contain only variables, constants, unary function symbols and the union operator (that is, no projection, intersection or terms with a function symbol of arity greater than one).

$$\begin{array}{lcl} e_l & ::= & x \mid a \mid f(e) \\ e_r & ::= & x \mid f(\overline{e}) \mid e \cup e' \mid e \cap e' \mid f^{-1}_{(k)}(e) \end{array}$$

For comparision, a *definite* set constraint [12] is a conjunction of inclusions $e_l \subseteq e_r$ between positive set expressions, where the set expressions e_r on the right hand side of $\subseteq$ are furthermore restricted to contain only variables, constants and function symbols and the intersection operator (*i.e.*, no projection or union).

We interpret set constraints over $2^{T_\Sigma^\infty}$, the domain of sets of infinite trees over the signature Σ. That is, variables denote sets of trees, and a valuation is a mapping $\alpha : \mathsf{Var} \to 2^{T_\Sigma^\infty}$. Tree constructors are interpreted as functions over sets of trees: the constant a is interpreted as $\{a\}$, and the function symbol f is interpreted as the function which maps sets $S_1, \ldots, S_n$ to the set

$$f(S_1, \ldots, S_n) \;=\; \{f(t_1, \ldots, t_n) \mid t_1 \in S_1, \ldots, t_n \in S_n\}\,.$$

The application of the projection operator for a function symbol f and the k-th argument position on a set S of trees is defined by

$$f^{-1}_{(k)}(S) = \{t \mid \exists t_1, \ldots t_n : \;\; t_k = t,\, f(t_1, \ldots, t_k, \ldots, t_n) \in S\}\,.$$

The symbols $\cup$, $\cap$ and $\subseteq$ are interpreted as usual. The union of set valuations $\bigcup_i \alpha_i$ on variables is the pointwise union on the images of all variables; *i.e.*, $(\bigcup_i \alpha_i)(x) = \bigcup_i \alpha_i(x)$.

We list three properties that are important in the proof of the soundness of abstraction in the next section. (1) The solutions of co-definite set constraints are closed under arbitrary union. (2) Every co-definite set constraint has a greatest solution if satisfiable. (3) Every co-definite set constraint without inclusions of the form $a \subseteq x$ is satisfiable.

The satisfiability problem for co-definite set constraints is DEXPTIME-complete (as for definite set constraints). The algorithm given in [2] computes the greatest solution of a satisfiable co-definite set constraint φ (written as $gSol(\varphi)$) in the form of a tree automaton.

φ_P. We will next describe the inference of a co-definite set constraint φ_P from a logic program P. (Given a cc program P, we note φ_P the constraint $\varphi_{\tilde{P}}$ inferred from the logic program $\tilde{P}$ corresponding to P.) We assume that the different clauses are renamed apart (if not, we apply α-renaming to quantified variables). We introduce a fresh variable z_t for each subterm t appearing in the formula and then define the constraint $\Phi(t \subseteq x)$ for a term t and a variable x by induction on the depth of t. (The constraints derived from the programs in the examples in Section 2 are syntactically simplified for better readability.)

$$\Phi(y \subseteq x) \;=\; y \subseteq x$$

and for $t = f(t_1, \ldots, t_n)$:

$$\begin{aligned}\Phi(t \subseteq x) \;=\; & z_t \subseteq x \;\wedge\; z_{t_1} \subseteq f^{-1}_{(1)}(z_t) \wedge \Phi(t_1 \subseteq z_{t_1}) \\ & \qquad \ldots \\ & \wedge\; z_{t_n} \subseteq f^{-1}_{(n)}(z_t) \wedge \Phi(t_n \subseteq z_{t_n})\end{aligned}$$

We define the constraint φ_P inferred from P as follows.

$$\varphi_P \quad \equiv \quad \bigwedge_{p \in \mathsf{Pred}} \; p = \bigcup_{i}^{n_p} t_i \;\wedge\; \bigwedge_{i}^{n_p} \bigwedge_{j}^{n_{i,p}} \Phi(t_{ij} \subseteq p_{ij})$$

Here, we treat both symbols $p \in \mathsf{Pred}$ and $x \in \mathsf{Var}$ as second-order variables which range over sets of trees. In the following, when we compare an interpretation ρ with a valuation σ of a set constraint, $\rho \subseteq \sigma$ means that $\rho(p) \subseteq \sigma(p)$ for all $p \in \mathsf{Pred}$.

Theorem 5.2 (Soundness of Abstraction) For any logic program P, the greatest model of P is smaller than the greatest solution of the co-definite set constraint derived from P; formally: $gm(P) \subseteq gSol(\varphi_P)$.

Proof. We first define an abstraction $T_P^\#$ of the T_P operator, and we prove that $gfp(T_P) \subseteq gfp(T_P^\#)$. Here, we extend valuations σ over trees to valuations α_σ over sets of trees by $\alpha(x) = \{\sigma(x)\}$. In the second part we show that $gfp(T_P^\#) \subseteq gSol(\varphi_P)$. That part exhibits an interesting connection between the fixed point equation and the set constraint (cf. also [13]).

1. $gfp(T_P) \subseteq gfp(T_P^\#)$. The T_P operator maps an interpretation ρ to another one $T_P(\rho)$ where, for all $p \in \mathsf{Pred}$,

$$\mathcal{T}_P(\rho)(p) \quad = \quad \left\{ t \in T_\Sigma \;\middle|\; \begin{array}{r} \exists \alpha : \mathsf{Var} \to T_\Sigma \; \exists i : \; t = \alpha(t_i), \\ T_\Sigma^\infty, \alpha \models \; t_{ij} \in \rho(p_{ij}) \end{array} \right\} . \qquad (3)$$

The greatest-model semantics and the greatest-fixpoint semantics of a program P coincide. That is, the greatest model of P's completion is the greatest fixpoint of the operator T_P, $gm(P) = gfp(T_P)$.

The $T_P^\#$ operator maps an interpretation ρ to another one $T_P^\#(\rho)$ where, for all $p \in \mathsf{Pred}$,

$$T_P^\#(\rho)(p) \;=\; \left\{ t \in T_\Sigma \;\middle|\; \begin{array}{r} \exists \sigma : \mathsf{Var} \to 2^{T_\Sigma^\infty}, \; \exists i : t \in \sigma(t_i), \\ 2^{T_\Sigma^\infty}, \sigma \models \; \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij}) \end{array} \right\} . \qquad (4)$$

Here, we use new variables x_{ij} as placeholders for p_{ij} (for better *legibility*: otherwise, p_{ij} would appear as a variable on which both, σ and ρ are applied). The variables $x \in \mathsf{Var}$ now range over sets of trees. We write $\mathcal{M}, \alpha \models F$ if the formula F is valid under the interpretation with the valuation α on the structure (with the domain) $\mathcal{M}$. The formula F above consists of co-definite set constraints in conjunction with inclusions between variables and constants $\rho(p_{ij})$ (interpreted as the corresponding set).

Let $\rho' = T_P(\rho)$ and $\rho'' = T_P^\#(\rho)$. Then $\rho'(p) \subseteq \rho''(p)$ holds for all $p \in \mathsf{Pred}$. This can be seen as follows. For every tree valuation α satisfying

the condition in the set comprehension for ρ', the set valuation σ_α defined by $\sigma(x) = \{\alpha(x)\}$ satisfies the condition in the set comprehension for ρ''. Clearly, $\sigma_\alpha(t_{ij}) \subseteq \rho(p_{ij})$; we replace the inclusion $t_{ij} \subseteq \rho(p_{ij})$ by the equivalent conjunction $t_{ij} = x_{ij} \wedge x_{ij} \subseteq \rho(p_{ij})$, and if σ_α satisfies the equality $t_{ij} = x_{ij}$ then also the weaker constraint $\Phi(t_{ij} \subseteq x_{ij})$.

Hence, $T_P^\#$ is indeed an abstraction of T_p, and, thus, $gfp(T_P) \subseteq gfp(T_P^\#)$. This concludes the first part of the proof.

2. $gfp(\boldsymbol{T_P^\#}) \subseteq gSol(\boldsymbol{\varphi_P})$**.** In order to show that $gfp(T_P^\#) \subseteq gSol(\varphi_P)$, we first reformulate the definition of $T_P^\#$ as follows.

$$T_P^\#(\rho)(p) \;=\; \bigcup_{\sigma:\mathsf{Var}\to 2^{T_\Sigma^\infty}} \bigcup_i \{\sigma(t_i) \mid 2^{T_\Sigma^\infty}, \sigma \models \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij})\}$$

Fix ρ and let $\rho'' = T_P^\#(\rho)$.

Next, we exploit the fact that the solutions of co-definite set constraints are closed under (arbitrary) union. This fact extends to formulas containing inclusions with set constants on the right hand side, such as $x_{ij} \subseteq \rho(p_{ij})$. Note that for all i, the formula in question is satisfiable. We obtain that

$$\rho''(p) \;=\; \bigcup_i \sigma_i(t_i) \quad \text{where} \quad \sigma_i \;=\; gSol(\bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij})).$$

Since all program variables are renamed apart, we have $\rho''(p) = \bigcup_i \sigma(t_i)$ where

$$\sigma \quad = \quad gSol(\bigwedge_i \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij})).$$

Thus, we have $\rho''(p) = \sigma(p)$ where

$$\sigma \quad = \quad gSol(p = \bigcup_i t_i \wedge \bigwedge_i \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij})).$$

Again, since all program variables are renamed apart,

$$\rho'' \quad = \quad gSol(\bigwedge_{p\in\mathsf{Pred}} p = \bigcup_i t_i \wedge \bigwedge_i \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij})).$$

Here, we equate the interpretation $\rho'' : \mathsf{Pred} \to 2^{T_\Sigma^\infty}$ with a valuation σ interpreting a formula with predicate symbols $p \in \mathsf{Pred}$ and tree variables $x \in \mathsf{Var}$ both ranging over sets of trees, and with constants of the form $\rho(p_{ij})$ standing for the *corresponding* sets. We omit any further formalization of this setting.

Let ρ_0 be any fixpoint of $T_P^\#$, *i.e.*, $T_P^\#(\rho_0) = \rho_0$. This means that ρ_0 is a solution (the greatest one, in fact) of

$$\bigwedge_{p\in\mathsf{Pred}} p = \bigcup_i t_i \wedge \bigwedge_i \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho_0(p_{ij}).$$

That is, ρ_0 is a solution of φ_P. Hence, ρ_0 is smaller than the greatest solution of φ_P. This is true in particular if ρ_0 is chosen as the *greatest* fixpoint of $T_P^\#$. This concludes the second part of the proof. □

Discussion: constraint logic programs. We might formulate the above set-based analysis in the setting of constraint logic programs.

$$P \;\equiv\; \bigwedge_{p \in \mathsf{Pred}} \forall x \;\; p(x) \leftrightarrow \bigvee_{i=1}^{n_p} \exists_{-x}(x = x_i \wedge x_i = t_i \wedge \bigwedge_{j=1}^{n_{i,p}} p_{ij}(x_{ij}) \wedge t_{ij} = x_{ij} \;)$$

$$\varphi_P \;\equiv\; \bigwedge_{p \in \mathsf{Pred}} \; p \subseteq x \wedge x = \bigcup_{i=1}^{n_p} x_i \wedge x_i = t_i \wedge \bigwedge_{j=1}^{n_{i,p}} x_{ij} \subseteq p_{ij} \wedge \Phi(t_{ij} = x_{ij}) \;)$$

We see that disjunction translates to union ("$x = \bigcup_{i=1}^{n_p} x_i$"). Moreover, the set constraint derived from the constraint $x_i = t_i$ is syntactically the same, whereas the set constraint $\Phi(t_{ij} \subseteq x_{ij})$ derived from the constraint $t_{ij} = x_{ij}$ is of a more complicated form (the reason is, intuitively, that x_i belongs to the head and the x_{ij}'s belong to the body).

The only two properties required in the analysis framework are: (1) a solution α of a constraint C can be "lifted" to a solution σ_α of the set constraint $\Phi(C)$ by setting $\sigma_\alpha(x) = \{\alpha(x)\}$, and (2) solutions of set constraints are closed under arbitrary unions.

Operational interpretation. We now summarize the operational interpretation of the result that is computed by the set-based analysis outlined above.

Theorem 5.3 (Set-based analysis of finite failure) The query $p(x)$ is finitely failed in every fair execution of the reactive logic program P if the value of p in the greatest solution *over sets of infinite trees* of the co-definite set constraint φ_P derived from P is the empty set; *i.e.*, for all predicates $p \in \mathsf{Pred}$, if $gSol(\varphi_P)(p) = \emptyset$ then $p(x) \in FF(P)$.

Proof. We combine Theorems 5.2 and 3.1. □

Theorem 5.4 (Set-based error diagnosis of cc programs)
The procedure p in the cc program P finitely fails or deadlocks if the value of p in the greatest solution *over sets of infinite trees* of the co-definite set constraint φ_P derived from P is the empty set.

Proof. We combine Theorem 5.3 and Proposition 4.2. □

Types. Using Theorems 3.1 and 5.2, the statements in the two theorems above can be made more precise (but then maybe less succinct). Given the greatest solution of φ_P, the value of the variable p describes an approximation of the set of argument terms t for p in non-erroneous queries in the following way. If the set of ground instances of the query $p(t)$ does not intersect with the value of p, then the query $p(t)$ will finitely fail (inevitably fail or deadlock, respectively). In this sense, the value of the variable p in the greatest solution of φ_P denotes a *type* for the arguments of the procedure p.

6 Related Work

Previous work set-based analysis for logic programming has considered the least model semantics only (see, e.g. [20, 12, 9, 1, 10]). Mishra's analysis [20] is often cited as the historically first one here. A comparison with our analysis sheds a new light on this.

Mishra uses a class of set constraints with a non-standard interpretation over non-empty *path-closed* sets of finite trees to approximate the least model of a logic program. Set constraints over path-closed sets share an important property with co-definite set constraints, which is: every satisfiable constraint has a greatest solution. (This property holds over path-closed sets even if n-ary constructor terms are allowed left of the inclusion. For example, the constraint $f(x,y) \subseteq f(a,a) \cup f(b,b)$ has a greatest model over path-closed sets (which assigns both variables x and y the set $\{a,b\}$) while it has two maximal but incomparable ones over the standard set domain. Therefore, constructor terms with an arity greater than 2 have been excluded from co-definite set constraints [2].)

Heintze and Jaffar [12] use the class of *definite* set constraints which are dual to co-definite ones in the sense that they have a least solution if satisfiable. They derive a definite set constraint ψ_P from a logic program P such that its least model is safely approximated by every solution of ψ_P, and best by the least solution. The values in a solution of ψ_P describe supersets of the possible runtime values in all successfully terminating executions.

Heintze and Jaffar [14] have shown that Mishra's analysis is less accurate than theirs in two ways, due to the choice of the greatest solution and due to the choice of the non-standard interpretation, respectively. The proof of the soundness of our abstraction carries readily over if we take Mishra's set constraints instead of co-definite ones. This means that Mishra's approximation is so weak that it even approximates the greatest model. Mishra was interested in terminating executions and hence considered logic programs with respect to the least model semantics. He proves that "$p(x)$ will never succeed" if p is analysed to be necessarily empty. For the case of *finite* trees, this is indeed all one can show. Our remarks above imply that one can strengthen this consequence by inferring that "$p(x)$ will always fail", provided that one takes non-empty path-closed sets of *infinite* instead of finite trees.

For the analysis of concurrent constraint programs, various techniques based on abstract interpretation habe been used (see, *e.g.*, [8]) but none are set-based. A first formal calculus for (partial) correctness of cc programs is developed in [7]. The proof methods there are more powerful than ours but not automatic. A thorough study of abstract diagnosis frameworks is given in [5]; for now, we have to leave open their relation with this work. The necessity to consider greatest-fixed point semantics for the analysis of reactive systems has been observed by other authors and in the context of different programming paradigms (see, *e.g.*, [6, 11]). To our knowledge,

this is the first application of set-based analysis to reactive programs. The application of the set-based analysis presented here, in combination with the one of [13], to the verification of general CTL properties of pushdown processes and other systems whose state can be modeled by *ground* queries of logic programs is explored in [3].

7 Conclusion

We have presented a set-based analysis of reactive logic programs (over infinite trees with the greatest-fixpoint semantics). This analysis is interesting in its own right, as a particular instance of static analysis, type inference or approximation of runtime values. It also helps to situate the existing fundamental studies of set-based analysis of (terminating) logic programs wrt. the least model semantics.

Using the characterization of finite failure of logic programs over infinite trees through the greatest model, and the connection between an error in cc programs and finite failure, we have applied our analysis to obtain an automated method for the static prediction of an important kind of runtime error (the disjunction of deadlock or failure) in concurrent constraint programs.

The realization of this framework for the Oz system, and its extension to reactive Oz programs with non-cc features such as cells and higher-order features is part of ongoing work. We have implemented a prototype version (with an incomplete constraint solver); we have used it in experiments which demonstrate its usefulness for finding bugs. More experiments are necessary to find the right balance between the efficiency and the accuracy of the diagnosis.

Finally, it has been an open question whether set-based analysis can be formulated also in the setting of constraint logic programming ever since the notion was coined in [13]. Our analysis might be a new starting point for reserach in that direction (see also Discussion at the end of Section 5).

References

[1] A. Aiken and T. Lakshman. Directional type checking of logic programs. In B. Le Charlier, editor, *Proc. Static Analysis Symposium, SAS'94*, volume 864 of *LNCS*, pages 43–60. Springer-Verlag, 1994.

[2] W. Charatonik and A. Podelski. Co-definite set constraints. In T. Nipkow, editor, *Proceedings of the 9th International Conference on Rewriting Techniques and Applications*, LNCS. Springer-Verlag, March-April 1998. To appear.

[3] W. Charatonik and A. Podelski. Set-based analysis of reactive infinite-state systems. In B. Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, March-April 1998. To appear.

[4] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, NY, 1978.

[5] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dam, editor, *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *LNCS*, pages 22–50. Springer-Verlag, June 1996.

[6] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proc. POPL '92*, pages 83–94. ACM Press, 1992.

[7] F. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages (POPL)*, pages 98–108. ACM Press, 1994.

[8] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Compositional Analysis for Concurrent Constraint Programming. In *Proceedings of the Eight Annual IEEE Symposium on Logic in Computer Science*, pages 210–221. IEEE Computer Society Press, 1993.

[9] T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, July 1991.

[10] J. P. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In *Proceedings of the Dagstuhl Workshop on Partial Evaluation*, pages 1–16. Springer-Verlag, February 1996.

[11] K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In *Proceedings of ICFP'97*, pages 38–51. ACM Press, 1997.

[12] N. Heintze and J. Jaffar. A decision procedure for a class of set constraints (extended abstract). In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51, 1990.

[13] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 197–209, January 1990.

[14] N. Heintze and J. Jaffar. Semantic types for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 141–156. MIT Press, 1992.

[15] J. Jaffar, M. Maher, K. Marriot, and P. Stuckey. The semantics of constraint logic programs. *J. Logic Programming*. To appear.

[16] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *The Journal of Logic Programming*, 19/20:503–582, May-July 1994.

[17] J. Jaffar and P. J. Stuckey. Semantics of infinite tree logic programming. *Theoretical Computer Science*, 46:141–158, 1986.

[18] J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Springer-Verlag, Berlin, Germany, second, extended edition, 1987.

[19] Programming Systems Lab, Universität des Saarlandes. The Oz System. Available through the WWW at `http://www.ps.uni-sb.de/oz`, 1997.

[20] P. Mishra. Towards a theory of types in Prolog. In *IEEE International Symposium on Logic Programming*, pages 289–298, 1984.

[21] M. Müller. *Type Analysis for a Higher-Order Concurrent Constraint Language*. PhD thesis, Universität des Saarlandes, Technische Fakultät, 66041 Saarbrücken, Germany, expected 1997.

[22] M. Nait-Abdullah. On the interpretation of infinite computations in logic programming. In J. Paredaens, editor, *Proceedings of the 11th Colloquium on Automata, Languages and Programming*, volume 172 of *LNCS*, pages 358–370. Springer-Verlag, July 1984.

[23] E. Palmgren. Denotational semantics of Constraint Logic Programming - a non-standard approach. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Constraint Programming*, volume 131 of *NATO ASI Series F: Computer and System Sciences*, pages 261–288. Springer-Verlag, Berlin, Germany, 1994.

[24] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, Lecture Notes in Computer Science 224, pages 510–584. Springer-Verlag, 1986.

[25] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *18^{th} Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, Jan. 1991.

[26] G. Smolka. The Oz Programming Model. In *Volume 1000 of Lecture Notes in Computer Science*. Springer-Verlag, 1995.