Forward-Tracking: A Technique for Searching Beyond Failure

Elena Marchiori\textsuperscript{1,3} \hspace{1cm} Massimo Marchiori\textsuperscript{2} \hspace{1cm} Joost N. Kok\textsuperscript{3}

\textsuperscript{1}CWI \\
P.O. Box 94079, 1090 GB Amsterdam (NL) \\
elsa@cwi.nl

\textsuperscript{2}University of Padova \\
via Belzoni 7, 35131 Padova (I) \\
max@math.unipd.it

\textsuperscript{3}University of Leiden \\
P.O. Box 9512, 2300 RA Leiden (NL) \\
joost@wi.leidenuniv.nl

Abstract

In many applications, such as decision support, negotiation, planning, scheduling, etc., one needs to express requirements that can only be partially satisfied. In order to express such requirements, we propose a technique called forward-tracking. Intuitively, forward-tracking is a kind of dual of chronological back-tracking: if a program globally fails to find a solution, then a new execution is started from a program point and state ‘forward’ in the computation tree. This search technique is applied to the specific paradigm of constraint logic programming, obtaining a powerful extension that preserves all the useful properties of the original scheme. We report on the successful practical application of forward-tracking to the evolutionary training of (constrained) neural networks.

1 Introduction

One of the most important search mechanism in automated reasoning is chronological back-tracking. Intuitively, the control can during the execution jump backwards in case of failure, in order to try other alternatives. If no alternative is left, then the execution stops. In many applications, however, like decision support, planning, and scheduling, one needs to express also some alternatives to (global) finite failure.

In this paper, we propose a novel technique, called forward-tracking, that allows one to formalize in a simple way problems as those described above, where in case of failure the constraints on the problem are relaxed in order to produce a satisfactory solution. Informally, this technique is based on the following idea: Some points of a program are selected as ‘forward-tracking points’ (ft-points); when a global failure occurs, the program is restarted from a new point ‘forward’ in the program, taking as state the ‘best’ value previously produced in one of the forward-tracking points. In order to define such a forward-tracking for a program, one needs a set of ft-points, a ‘jump function’ to restart the program from another point, and a ‘height ordering’ (to select the best value). Forward-tracking is useful in all the situations where we are interested in an optimal solution with respect to several requirements, and where we require that a satisfactory partial solution be given in case of failure. These situations include for instance conflicting requirements, over-constrained problems, partial knowledge problems, prioritized reasoning and so on.

In this paper we apply forward-tracking to the specific paradigm of constraint logic programming (cf. \cite{6}), a general scheme that allows one to express in a declarative way reasoning in the presence of constraints. We investigate semi-automatic and (almost) fully-automatic methods for computing forward-trackings. At the most automated level, the user can just use a constraint logic program, which is automatically executed with forward-tracking; in another almost automated level, the user has only to insert one additional predicate, named separation, in the locations where (s)he thinks there is a separation of concerns. Forward-tracking is computationally little expensive, since it can be implemented in such a way that: as far as time complexity is concerned, there is only a small (fixed-time) overhead each time a forward-tracking point is reached or a forward jump is performed; as far as space complexity is concerned, forward-tracking only needs a limited fixed-size amount of extra memory w.r.t. the original program. Forward-tracking allows to formalize complex problems that can be subject to various constraints, and cannot be easily tackled with usual techniques, in an elegant and modular way as follows. First, the various aspects of the problem are specified by means of modules, each module describing a particular aspect. Next, the forward-tracking technique is used for combining the modules with respect to the specific instance of the problem considered, filtering out possible inconsistencies, in order to obtain satisfactory solutions. This design approach is illustrated in a practical application (Section 6), namely the
training of constrained neural networks.

We turn now to the relation of our proposal to previous work. Related work includes Hierarchical Constraint Solving and the corresponding Hierarchical Constraint Logic Programming (HCLP) systems (see e.g. [1, 9]). In these schemes, constraints hierarchies are used in order to express preferences as well as requirements among constraints. Intuitively, the constraints used in the problem are labeled by means of preferences, which express the strengths of the constraints. During the execution of the program, non-required constraints are accumulated and they are solved only after the execution of the query is successfully terminated, selecting a method that depends on the domain and on the technique used for comparing solutions. The incremental version of this technique ([9]) tries to satisfy constraints as programs, i.e., when the information that we want to discard the execution of the query is successfully terminated, select a method that depends on the domain and on the technique used for comparing solutions. The incremental version of this technique ([9]) tries to satisfy constraints as soon as they appear, requiring a special backward algorithm. The main differences of forward-tracking with the above approach are: Forward-tracking does not depend on the particular constraint solver employed; it is applicable also to logic programs, i.e., when the information that we want to discard in case of finite failure is described by some predicates; it is built ‘on top’ of the program, and does not need, like the previous paradigms, a specific constraint solver to be built from scratch. Moreover, forward-tracking is transparent with respect to the original program, in the sense that it is only activated when the program globally fails; in all the cases when the original program gives a solution, forward-tracking does not interfere. Thus forward-tracking can be optionally activated on request, when the original program fails. Finally, forward-tracking, in its dynamic version (Subsection 5), allows to give preferences among constraints that are created during the execution of the program (i.e. ‘temporal’ priorities), which is not possible in the hierarchical constraint solving paradigm.

Other related work concerns the relaxation of constraints in case of conflicts (e.g. [3, 2]). Recently, in order to discard failed subproblems in a constraint satisfaction problem, [4] introduced a method that dynamically discards failed subproblems during forward checking search. This techniques can be successfully integrated with forward-tracking for constraint logic programming since the latter, as said, is a search technique that works on top of a constraint solver.

2 Searching Beyond Failure

In this section we explain the technique of forward-tracking. In order to focus more on the technique than on the programming paradigm where it is employed, we shall describe in a rather informal way the operational model. The incorporation of the technique into a specific programming paradigm, namely Constraint Logic Programming, is examined in detail in the next section.

The operational semantics of a program language can be defined in terms of a transition system with rules of the form (point, state) → (point′, state′) if conditions, meaning that if the execution reaches a certain (program) point with a certain state, and if the conditions are satisfied, then the execution goes to point' with state state'. Usually, an initial (start) and a final (end) point are used to represent input and output, respectively. Given an input state input, the set of computations is considered, obtained using the rules of the transition system starting from the pair (start, input). Then the operational semantics of the program is defined, w.r.t. an input, by abstracting the set of computations from (start, input), according to the information one is interested in. Finite computations ending in (end, s) are called successful and s is called output state. Instead, a program fails (w.r.t. an input state) if it does not compute any output state, while it finitely fails if it fails and all its computations are finite.

Forward-tracking allows to deal with finite failure, by allowing the computation to restart at a suitable point, from a suitable state among those that have been produced during the computations. In order to define a forward-tracking, a suitable subset FTP of points, called forward-tracking points (ft-points), is selected, representing the points from which one can jump forward in case of failure. Moreover, a height partial ordering <_H on pairs (ζ, state) consisting of a point and a state is used, in order to choose a proper pair in case of finite failure. Finally, a jump function J on ft-points is employed, that specifies how to jump forward. In the sequel, P denotes a program, PP and FTP the set of its points and ft-points, respectively. Moreover, S denotes the set of states. We have the following definitions.

Definition 2.1 A jump function, denoted by J, is a function FTP → FTP. An height ordering (w.r.t. FTP), denoted by <_H, is a partial order on FTP × S. A forward-tracking FT for P, is a pair (J, <_H), where J and <_H are a jump function and a height ordering on FTP, respectively. The triple (P, J, <_H) is called forward-tracking program (ft-program for short).

The operational semantics of a ft-program can be roughly explained as follows. The execution keeps track of the combined information on ft-points and states, by means of pairs consisting of a ft-point and a state, indicating that the control has passed through that ft-point in that state. During the execution a set of best pairs is accumulated, containing the maximal (w.r.t. <_H) pairs among those that have been produced from the begin of the execution till that moment. When the program finitely fails, and the control has passed through some ft-point, i.e. the set of best pairs is not empty, one best pair is selected, say (ζ, state), and a new execution is started from the ft-point obtained by applying J to ζ, in the state state. Observe that there can be more than one way to re-start the program, depending on the maximal element that is selected.
In order to work properly, a forward-tracking has to satisfy some correctness properties (otherwise, we would have a kind of unrestricted ‘goto’ feature).

**Definition 2.2** A forward-tracking \((J, <_J)\) is correct if the following conditions hold:
1. For every \(\zeta, \eta \in \text{FTP}\), if \(J(\zeta) = \eta\) then every successful computation of \(P\) that reaches \(\zeta\) passes also through \(\eta\).
2. If \(P\) starting from \((\text{start}, s)\) successfully computes an output, then \((P, J, <_J)\) starting from \((\text{start}, s)\) computes the same output.
3. If \((P, J, <_J)\) with input input finitely fails, then \(P\) with input input finitely fails.

The first condition ensures that the flow of control of the original program is respected: in other words, that the jump should be a ‘forward’ jump, and not a ‘backward’ jump. The other two conditions ensure that the ft-program behaves at least as well as the corresponding program, i.e., it does not lose successful computations (condition 2.); and it does not add finite failure computations (condition 3.).

## 3 Specifying Complex Problems

In this section we present a methodology based on the forward-tracking technique for the treatment of complex problems.

In general, a problem \(P\) is made up by several factors, say \(P_1, \ldots, P_k\). These sub-problems might be tightly coupled to each other by interdependencies, i.e., not forming a ‘modular decomposition’ of the general task: in order to get the original problem from the sub-problems, one has to face complex interactions among them. Thus, \(P_1, \ldots, P_k\) can form a kind of ‘lossy’ modular decomposition of the problem. One can ‘modularize’ \(P\) using forward-tracking in the following way. First, separate modules for \(P_1, \ldots, P_k\) are developed, say \(M_1, \ldots, M_k\). Next, possible conflicts among \(P_1, \ldots, P_k\) are ‘solved’ by means of forward-tracking, by employing the separation given by the modules. This can be realized by inserting separation points among the modules, as illustrated in the picture below by big gray circles:

This way each time a module is invoked, it starts solving the specific problem. When it cannot progress further, the control forward-tracks to the next module, and all the work done in the module is preserved. After the automatic calculus of forward-tracking, using e.g. the techniques of Section 5, a situation as the one pictured below is reached, where the small black circles denote the ft-points:

![Diagram showing the process of forward-tracking and modularization](image)

where a possible execution is

Note that modules are applied in a certain order, this way expressing preferences among them: modules appearing prior in the chain will have higher priority than modules appearing later.

## 4 Forward-Tracking for CLP

In this section we introduce a formalization of forward-tracking for the CLP paradigm (cf. [6]), and provide a sufficient condition for the correctness of a forward-tracking.

A domain \(D\) of constraints is considered, where a constraint \(c\) is a (first-order) formula built from primitive constraints. The states are the consistent constraints, denoted by \(c, d\). In the sequel, \(A\) (possibly subscripted) denotes an atom as well as a constraint; for an atom \(A\), \(\text{pred}(A)\) indicates its predicate symbol. Queries (i.e., sequences of atoms and constraints) and clauses are supposed to be decorated with distinct program points, denoted by \(\zeta, \eta, \nu\). For a clause \(H \leftarrow \zeta_0 A_1 \ldots \zeta_i A_i \ldots \zeta_k A_k\), \(\zeta_i\) is the calling point of \(A_i\), denoted by \(\text{cp}(A_i)\), and \(\text{atom}(\zeta_i)\) indicates \(A_i\). A constraint logic program (clp) is a set of clauses augmented with a query \(Q\). The points \(\text{start}\) and \(\text{end}\) are the leftmost and rightmost point of \(Q\), respectively.

In the operational model we are going to describe, the Prolog left-to-right selection rule is assumed. A transition system for ft-clp’s is built on the top of an operational semantics for clp’s, that describes also information on ft-points, and is defined w.r.t. an height ordering \(<_H\).

We consider a transition system for clp’s, defined on two kinds of tuples: either \((Q, c, S, \phi)\) or \((\text{fail}, S, \phi)\), where \(S\) is a set of triples of the form \((\xi, d, \psi)\), with \(\xi\) an ft-point, \(d\) a state, and where \(\psi\) and \(\phi\) are functions mapping points into functions that rename the program variables. \(S\) and \(\phi\) are used for keeping track of the ft-points encountered during the computation together with the corresponding states, and for each ft-point, of the most recent renamings of the variables of the program clause where it occurs, respectively. The latter information, described by \(\phi\), is needed when a jump forward is made. In this case, \(\phi\) is used to relate the variables of the clause containing the point where the computation has jumped and the variables occurring in the state used to restart
the computation from that point. There are four transition rules. The two transition rules for steps yielding failure are the standard, except that here the information on \( S \) and \( \phi \) is kept, i.e., \( \langle Q, c, S, \phi \rangle \rightarrow \langle \text{fail}, S, \phi \rangle \). The other two rules treat successful computation steps as follows, where \( cp(Q) \) denotes the leftmost calling point of \( Q \).

**Resolution:** \( \langle \pi(Q), c, S, \phi \rangle \rightarrow \langle R, c', S', \phi' \rangle \), where \( \pi(T) \leftarrow R \) is a renamed apart clause of \( P \), say by means of the function \( p, c' = (c \land \pi = T) \) is consistent, \( S' \) consists of the maximal triples of \( S \cup \{cp(R), c', \phi'\} \), and \( \phi' \) is equal to \( \phi \) except for the points that are in \( R \) that are mapped into \( p \). Here and in the sequel maximal is intended w.r.t. \( \prec_H \), when projecting away the third component of the elements.

**Constraint:** \( \langle c, Q, c, S, \phi \rangle \rightarrow \langle Q, c \land d, S', \phi \rangle \), where \( S' \) consists of the maximal triples of \( S \cup \{cp(Q), c \land d, \phi\} \).

Observe that in the last two rules we could have chosen \( S' \) to be equal to \( S \cup \{cp(R), c', \phi'\} \). However, for efficiency reasons, we have considered a (in general) smaller set.

The operational semantics for a clp \( P \cup \{Q\} \) w.r.t. \( \prec_H \) is defined as follows. Below, \( I \) denotes the function mapping each point into the identity renaming \( c \rightarrow \ast \) denotes the reflexive and transitive closure of \( \rightarrow \), and \( \neq \) denotes the empty query.

**Definition 4.1** The (input-output) operational semantics \( O(P \cup \{Q\}) \) of \( P \cup \{Q\} \) (w.r.t. \( \prec_H \)) is the set \( SS(Q) \cup FS(Q) \), where \( SS(Q) \) is the set of pairs \( (c, c') \) s.t. \( \langle Q, c, \{\text{start}, c, I\}, I \rangle \rightarrow (\square, c', S, \phi) \) for some \( S, \phi \), and \( FS(Q) \) is the set of tuples \( (c, S, \phi) \) s.t. \( \langle Q, c, \{\text{start}, c, I\}, I \rangle \rightarrow (\square, S, \phi) \).

It is not difficult to check that the standard input-output semantics of (ideal) clp’s with left-to-right selection rule can be obtained as an abstraction of \( O(P \cup \{Q\}) \), by adding to \( SS(Q) \) the set of pairs \( (c, \text{fail}) \) with \( (c, S, \phi) \in FS(Q) \).

A derivation is defined in the usual way, as a sequence of elementary steps obtained using the transition system. The (standard) notion of finite failure is defined as follows. \( \langle Q, c \rangle \) finitely fails if every its derivation is finite and \( SS(Q) \cap \{(c, c') \mid c' \in D\} = \emptyset \).

The transition system for ft-clp’s uses \( O(P \cup \{Q\}) \) to specify the applicability conditions of the rules. It is defined either on tuples of the form \( \langle Q, c \rangle \) or \( \langle \text{fail} \rangle \). It consists of the following rules:

**Success:** \( \langle Q, c \rangle \rightarrow (\square, c' \land c) \) if \( (c, c') \in SS(Q) \).

**Forward-tracking:** \( \langle Q, c \rangle \rightarrow (Q', c' \land c) \) if \( (Q, c) \) finitely fails, \( (c, c', \phi) \) is maximal among the elements of all the sets \( S \) such that \( (c, S, R) \) is in \( FS(Q) \); \( R \) is the clause-body (or the query) containing \( J(c) \), \( p = \phi(J(c)) \) and \( Q' \) is the suffix of \( RP \) starting at \( J(c) \).

**Failure:** \( \langle Q, c \rangle \rightarrow (\text{fail}) \) if \( (Q, c) \) finitely fails and there is no maximal element among the elements of all the sets \( S \) s.t. \( (c, S, R) \) is in \( FS(Q) \).

The input-output semantics of a ft-clp can be defined as follows.

**Definition 4.2** The (input-output) operational semantics \( O_P(U\{Q\}) \) of \( P \cup \{Q\} \) is the set \( SS(Q, FT) \cup FS(Q, FT) \), where \( SS(Q, FT) \) is the set of pairs \( (c, c') \) s.t. \( \langle Q, c \rangle \rightarrow (\square, c', \phi) \) and \( FS(Q, FT) \) is the set of pairs \( (c, \text{fail}) \) s.t. \( \langle Q, c \rangle \rightarrow (\text{fail}) \).

**Example 4.3** A Hamiltonian path of a graph is an acyclic path containing all the nodes of the graph. The following (fragment of the) general logic program hamiltonian defines hamiltonian paths.

\[
\text{ham}(G, P) \leftarrow \text{path}(N1, N2, G, P) \land \text{cov}(P, G).
\]

\[
\text{cov}(P, G) \leftarrow \text{notcov}(P, G).
\]

\[
\text{notcov}(P, G) \leftarrow \text{node}(X, G) \land \text{notmember}(X, P).
\]

The relation \( \text{ham}(g, p) \) is specified in terms of \( \text{path} \) and \( \text{cov} \): it is true if \( p \) is an acyclic path of \( g \) that covers all its nodes. Acyclic paths of a graph are specified by means of the predicate \( \text{path} \).

We assume that negation by (finite) failure is used for resolving negated atoms. Using a forward-tracking, the behaviour of \( \text{hamiltonian} \) can be ‘relaxed’ in order to obtain useful answers also when there is no hamiltonian path. Consider the order \( \preceq \) on substitutions s.t. \( \alpha \preceq \beta \) if \( \beta = \alpha \gamma \) for some \( \gamma \). Define \( J(\xi) = \text{end} \), and \( (\xi, \alpha) \prec_H (\xi, \beta) \) if \( \alpha < \beta \). Then, the ft-program \( \text{hamiltonian} \cup \{\text{ham}(g, X)\} \) produces answers also for those graphs containing no hamiltonian paths, where \( \text{ham}(g, X) \) will bind \( X \) to an acyclic path of \( g \) of maximal length. We will show in the next section that the resulting ft-program is ‘correct’.

**Correct Forward-Trackings**

We introduced now a sufficient criterion for the correctness of the forward-tracking for ft-clp’s. To this aim, we assume that a jump function must respect a certain forward ordering \( \prec_F \). This ordering imposes reasonable conditions on possible jumps, preventing the introduction of cycles in the derivations, and ensuring that the flow of control of the original program is ‘respected’: a jump from \( \zeta \) to \( \eta \) is allowed only if, in order to reach \( \eta \) the control (in the original clp) has first to reach a point that is in the same clause of \( \eta \) and occurring to its left, and then to reach \( \zeta \). For instance, consider the program clauses \( p \leftarrow q, r, s \leftarrow c, c \leftarrow \ldots \). One can jump from \( cp(q) \) to \( cp(r) \), but not from \( cp(q) \) to \( cp(c) \), because the corresponding clauses are completely unrelated. The effect of the latter kind of jumps is similar to that of the goto-like statements in the imperative paradigm, and therefore they are discarded.

In order to formalize the notion of forward ordering, we use the following dependency relation \( \text{dep} \) on points. In the sequel, we write \( \eta \) is (immediately) to the right/left of \( \zeta \) as a shorthand for \( \zeta \text{ and } \eta \) occur in the same clause and \( \eta \) is (immediately) to the right/left of \( \zeta \).
Definition 4.4 \( \text{DEP} \) is the transitive closure of the relation \( \text{dep} \) on program points defined by \((\zeta, \eta) \in \text{dep}\) if either \(\text{atom}(\eta)\) and the head of the clause containing \(\zeta\) have the same predicate symbol, or \(\text{atom}(\eta)\) is a constraint and \(\eta\) is immediately to the right of \(\zeta\).

Example 4.5 Consider the program clauses
\[ p \leftarrow c_1 q c r, \quad q \leftarrow c_2 s, \quad r \leftarrow c_3 s. \]
Then \( \text{DEP} = \{(c_3, c_1), (c_4, c_2), (c_5, c_1), (c_4, c_3), (c_5, c_2)\}. \)

The following definition of safe pair of points formalizes when a jump respects the control flow of the program. For a subset \( T \) of points, we say that \( T \) has a maximum \( \nu \) if for every \( \zeta \in T \) and every \( \zeta \neq \nu \) occurring in \( T \) we have that \((\nu, \zeta) \in \text{DEP}\) but \((\zeta, \nu) \notin \text{DEP}\).

Definition 4.6 A pair \((\zeta, \eta)\) of points is safe if there exists a set \( C \) of points s.t.:
1. \( \zeta \in C \);
2. \( C \) has a maximum, say \( \nu \);
3. for each \( \zeta \in C \), if \( \zeta \neq \nu \) and \((\zeta, \eta) \in \text{DEP}\) then \( \eta \in C \);
4. \( \eta \) is to the right of \( \nu \).

Observe that if \( \eta \) is to the right of \( \zeta \) then \((\zeta, \eta)\) is safe (take \(\zeta = \{\zeta\}\)). This formalizes that it is possible to jump from a point to another point of the same query in the direction of the flow of control, i.e. from left to right, unless the jump introduces a cycle. This latter requirement is expressed by condition 2. of the following definition of forward ordering.

Definition 4.7 The forward ordering \( \preceq \) is the greatest partial ordering \( \leq \) on points such that if \( \zeta \preceq \eta \) then
1. \((\zeta, \eta)\) is safe;
2. \((\zeta, \eta) \notin \text{DEP}\);
3. if \( \zeta = \text{end} \) then \( \eta = \text{end} \).

We illustrate by an example how condition 2. prevents the introduction of cycles.

Example 4.8 Consider the program clauses
\[ p \leftarrow c_1 q c r, \quad q \leftarrow c_2 s, \quad r \leftarrow c_3 s. \]
If we would not have condition 2. in the definition of forward ordering, then the set \( \{(c_3, c_1), (c_4, c_2)\} \) would be a legal ordering. But following the intended interpretation, this ordering yields the infinite computation \( p, q, r, p, \ldots \), while in the original program all derivations are finite.

We say that a ft-clp \( (P \cup \{Q\}, J, \prec_h) \) respects the forward ordering \( \preceq \) if the jump function \( J \) is such that \( \zeta \preceq \eta \) \( J(\zeta) \) (provided \( \zeta \neq \text{end} \)). The following result holds.

Theorem 4.9 If a ft-clp respects the forward ordering, then it is correct.

As far as the first condition of correct forward-tracking is concerned, an ft-clp respecting \( \preceq \) satisfies even more:

Lemma 4.10 Let \( (P \cup \{Q\}, J, \prec_h) \) be a ft-clp respecting the forward ordering. Let \( \zeta, \eta \in \text{FTP} \) and suppose that \( J(\zeta) = \eta \). Every computation of \( Q \) that reaches \( \zeta \) passes also through a point to the left of \( \eta \).

The above lemma is important, because it ensures that when a jump is performed from a given point equipped with a state, to another point, the state can be safely attached to the new point, because it describes information on (some renaming of) the variables of the clause containing that point.

Another important property of forward-trackings respecting \( \prec \) is that they behave well w.r.t. termination, in the sense that they preserve strong termination. Recall that a (ft-)clp \( P \) is strongly terminating if every \( Q \) has only finite computations. Moreover, passing through ft-points eliminates finite failure. Thus, the following results hold.

Theorem 4.11 Suppose \( P \) is strongly terminating. For every query \( Q \), if \( \text{FTP} \) is a forward-tracking for \( P \cup \{Q\} \) respecting \( \prec \) then \( (P \cup \{Q\}, \text{FTP}) \) has only finite computations.

Proposition 4.12 Every computation of a ft-clp that passes through a ft-point is either non-terminating or successful.

Example 4.13 It is easy to check that the ft-program \( (\text{hamiltonian} \cup \{\text{ham}(g, X)\}, J, \prec_h) \) given in Example 4.3 respects the forward ordering, hence by Theorem 4.9 it is correct.

5 Computing Forward-Trackings

In this section, we investigate various techniques for constructing forward trackings. First, we assume a jump function given, and introduce automatic methods for generating a corresponding height ordering. Next, we investigate automatic techniques for constructing both the jump function and the height ordering.

Semi-Automatic Techniques

Assume a jump function \( J \) respecting the forward ordering is given. Then one can automatically derive a forward-tracking as follows. Consider the partial ordering \( \prec \) on \( \text{FTP} \) s.t.:
\[ \forall \zeta, \eta \in \text{FTP}, \forall c, c', (\zeta, c) \prec (\eta, c') \Rightarrow \zeta \prec \eta. \]

Then \( (J, \prec) \) is a correct forward-tracking. An interesting feature of forward-trackings having \( \prec \) as height ordering is that they do not depend on the constraint solver, and therefore they can be "universally" implemented, in the sense that, under reasonable assumptions, there is an automatic compilation of ft-clp’s into clp’s. This way, every ft-clp written in a particular CLP system can be (automatically) implemented using the same system.

Automatic Techniques
We have seen how forward-trackings can be derived semi-automa-
ically, where the user provides a \( J \) function, without
bothering about the height ordering. However, this may
still be a tedious task, especially for large and complex
programs. In this subsection, we present a technique for
deriving forward-trackings almost automatically.

The idea is that the user is only concerned with the choice
of a suitable sequence of pp's, indicating those points that
separate different tasks, and defined as follows.

**Definition 5.1** A sequence of points \( S = v_1, \ldots, v_n \) is a separa-

ation if \( v_1 < _s v_2 < _s \ldots < _s v_n = \text{end} \).

From a separation \( S = v_1, \ldots, v_n \) one can derive auto-
matically a jump function \( J_S \) as follows. Consider the set
\( \text{FTP} \) of ft-points that can reach (via \( < _s \)) a separation point,

i.e., the points \( \zeta \) such that \( \zeta < _s v_i \) (for some \( i \in [1, n-1] \)).
From each of these points we can jump to the nearest (w.r.t.
\( < _s \)) separation point, by means of a function \( i \) characterized
as follows.

**Proposition 5.2** For every \( \zeta \in \text{FTP} \) there is a unique num-
ber \( i(\zeta) \in [1, n] \) such that \( \zeta < _s v_i(\zeta) \), and if \( i(\zeta) > 1 \) then
\( \zeta < _s v_{i(\zeta)-1} \).

Then \( J_S \) is defined by \( \Upsilon(\zeta) \in \text{FTP}, J_S(\zeta) = v_i(\zeta) \). It can be
proven that \( J_S \) is a jump function respecting the forward
ordering. Therefore, one can utilize the semi-automatic
approach introduced in the previous subsection to automat-
ically obtain a corresponding \( < _s \) height ordering.

The specification of a separation can be integrated in the
syntax of the program using a distinguished predicate name
"separation". This predicate is transparent in the sense that
it doesn't affect the normal execution of the CLP (i.e., it
is defined by means of the fact \( \text{separation} \rightarrow \)). Its role is
only to indicate the corresponding separation point. This
way, the user indicates the separation points of the program
by inserting the "separation" predicates in the corresponding
points. A syntactical checker can be designed that tests if the
so far inserted separation predicates form a separation. Since
the notion of separation relies on the syntactically defined
ordering \( < _s \), a facility that visually specifies what positions
that are allowed for the next insertion of a separation can be
incorporated into the checker.

Note that one can perform forward-tracking also in a
completely automatic way (i.e. not requiring the user to do
a separation), by using the trivial separation consisting only
of the \( \text{end} \) point. This means that the final output will be
the 'best' (w.r.t. the height ordering) result produced by the
program during the execution (cf. Example 4.3).

**Dynamic Forward-Tracking**

There can be situations where we know that some
progress is made, but it is hard to recover this information
from the state. This is due to the fact that the height ordering
has to be provided *statically*. In this section, we propose a
technique for the dynamic construction of forward-tracking,
i.e. the height ordering is built *dynamically* during the exe-
cution of the program.

An interface to the user can be defined, similar to the
"separation" predicate of the previous section, by means of
a distinguished predicate "progress". In order to assist the
inference of the height ordering, this predicate is inserted in
those points where the execution is considered (by the user)
to progress. Like "separation", "progress" is transparent, so it
doesn't affect the normal execution of the CLP. Using the in-
formation provided by "progress", a dynamic height ordering
is automatically defined by extending \( < _s \) as follows. Sup-
pose a forward-tracking has to be activated. This means that
an execution of the (original) program has failed. Suppose
that the execution passes first through a point \( \zeta \) in a state \( c \);
that the "progress" predicate is also executed; and then that the
execution passes through a point \( \eta \) in a state \( c' \). We add to
the ordering the condition that \((\eta, c')\) is greater than \((\zeta, c)\),
provided this doesn't clash with the existing ordering. That
is, when a "progress" predicate is encountered all the subse-
sequently obtained pairs (program point, state) are considered
to be higher than those previously obtained.

This execution model is sound since it can be proved that
at any stage of the execution such dynamically built ordering
is an height ordering that, together with \( J \), forms a correct
forward-tracking.

An automatic implementation of this dynamic approach
can be provided by just inserting progress predicates before
each point in \( \text{FTP} \) (the heuristic is that the more predicates
are invoked, the more work is done): We call dynamic
"separation" via \( < _s \) the automatic generation of a forward-
tracking from a separation by applying this approach to
\( < _s \). At first sight such dynamic generation of the height
ordering seems to be *extremely* expensive. However, this is
not the case. For instance, a special global variable can be
used that memorizes only the best-so-far pairs (relatively to
"progress"): these are updated only when a "progress" predicate
is reached. Also, relative information on the 'progress'
ordering is maintained as a counter in the constraint store
(and again, it is updated only when a "progress" predicate
is reached): this way, backtracking automatically does the
job of "undoing" progress information. The neat effect of
such an implementation is that there is only a small (fixed-
time) overhead when a "progress" predicate is executed, and
then the selection of the maximal program point w.r.t. the
height ordering is performed without overhead (since we
have available the 'best-so-far' result, and so actually we
are not maintaining the whole ordering but only the best
couple to select for forward-tracking).

**Refining the Height Ordering**

The automatic height ordering \( < _s \) used so far can be safely
replaced in all the previous techniques by a more sophisti-
cated ordering, provided that the new ordering does not conflict with \( \mathcal{A} \). The height ordering \( \mathcal{A} \) depends only on the points (i.e., on the program structure). We investigate now an extension of \( \mathcal{A} \) obtained by exploiting some semantic information on the states associated with points. Somehow, one should define an ordering on states too, that provides a measure of the computational information that is represented by a constraint. To infer such information, one has to rely on the particular CLP system. Therefore, we assume that the considered CLP system allows to compute the character length of the representation of the current state (this condition is satisfied by many CLP systems). For a state \( c \), let \(|c|\) denote its character length. An height ordering \( \mathcal{A} \) extending \( \mathcal{A} \) can be defined by:

\[
\forall \zeta, \eta \in FTP, \forall c, c': (\zeta, c) \mathrel{\mathcal{A}} (\eta, c') \iff
\] either \( \zeta \mathrel{\mathcal{A}} \eta \) or \( \zeta \) and \( \eta \) \( \mathcal{A} \)-incomparable and \(|c| < |c'|\).

As expected, we have that if \( J \) is a jump function respecting the forward ordering, then \((J, \mathcal{A})\) is a correct forward-tracking. This technique can be parameterized using different orderings on states. Here we have employed the ordering induced by the \( |\cdot| \) mapping. Other predefined orderings can be used, for instance the one induced by the map counting how many different variables are present in the state. As for \( \mathcal{A} \), \( \mathcal{A} \)-based ft-clp’s can be automatically compiled on the considered CLP system (under reasonable assumptions, and provided, as said, that the \( |\cdot| \) mapping can be written in it).

### 6 Constraining Neural Networks

In this section we illustrate an application of the forward-tracking technique to solve an interesting problem in the field of neural networks, namely the training of a constrained neural network.

Neural networks have been used in many applications, for example in planning, control, content-addressable memory, optimization, constraint satisfaction, and classification (see e.g. [5]). They are being promoted for their robustness, massive parallelism, and ability to learn. The training of a neural network is a major design step: Roughly, one has to find a set of weights that minimizes the neural network's error on an initial set of input-output examples called the training set. The standard method for that problem is a local gradient search method known as the back-propagation algorithm. Alternative approaches based on Genetic Algorithms (GA’s) have been proposed, which apply for instance to recurrent networks, or to networks with non-differentiable error criteria (for example due to non-differentiable transfer functions, error measures that use absolute values, bonuses for small weights etc.). However, there is a well-known problem with the application of genetic algorithms to neural networks, called the competing conventions problem. When one chooses a representation (in this case for a neural network), then it can be the case that the same individual has more than one representation. (i.e. many structurally different networks can nevertheless represent the same functional mapping). In order to solve the competing conventions problem, a novel approach has been introduced in [8]. The idea is to develop suitable constraints (via a CLP) that avoid multiple representations of individuals, and then performing an evolutionary training of the neural network by using only chromosomes satisfying these constraints. In [7] the study of regularities of the error function has revealed that single regularities are in some cases solvable: for example when the network has a single (hidden) layer. However, in order to cope with a generic network and with generic regularities of the error function, one has to face two assembling problems: composing layers, where the constraints generated for each layer are composed; and composing regularities, where the constraints generated for each regularity are composed. So, this is an example of hard and complex problem for which a decomposition in smaller factors is known (the regularities, the layers), but such a decomposition is not immediately useful from a modularization point of view.

In order to cope with the general, intractable case one can use the forward-tracking technique, following the methodology proposed in Section 3. We consider constrained networks, e.g. neural networks with shared weights, constraints on the weights - for example domain constraints for hardware implementation - etc. Moreover, other constraints are generated ensuring that in most cases each network is specified by exactly one chromosome. Thus the problem becomes a constrained optimization problem. The optimization criterion is to optimize the error of the network (usually this error includes a sum over all the training patterns of the network), and the constraints specify the domain constraints on the weights and those constraints used for avoiding the competing conventions problem. More precisely, a ft-clp is used, for producing constraints on the weights such that each network has in most cases a unique chromosome representation. This way, the CLP system also checks for the satisfiability of these constraints. Then the obtained constraints and the optimization function are given as input to a GA system, that searches for an optimal solution that satisfies the constraints. We have used the CLP system ECL/PS* (ECRC Common Logic Programming System) and the GA system GENOCOP [10] (GEneric algorithm for Numerical Optimization for Constrained Problems). Since the weights are real numbers, and they are constrained, a natural choice is to employ a system that can handle constraints and where the data are encoded using real numbers, instead of the original bit-encoding. The GENOCOP system satisfies both these requirements. Following the methodology in Section 3, we developed modules solving particular regularities. These modules have then been composed via the separation structure. For each layer of the network we built up a constraint generator solving the regularity, and
then merged these modules in a separation structure. The situation is illustrated below:

![Separation Structure Diagram]

This approach allows an easy and flexible modular control over the constraint generations: if the user is aware of some new specific regularity due to the particular activation function chosen, or due to symmetries in the data, etc., (s)he can safely add a submodule performing this specific task. This is also the case if (s)he knows that some cases will not occur, in which case some submodules which are not needed and that will maybe restrict too much the search space can safely be removed.

Also, the user is free to experiment with every combination of modules (the first choice could be for instance to give priority in the chain to modules according to the importance of the corresponding regularity, i.e. how much it affects the search space; then, some other combinations could also be tested).

To this respect, empirical tests showed that forward-tracking behaves very well, in the sense that the performance remains practically unaltered when different orderings are chosen for the modules in the chain (i.e. using different orderings leads to minimal differences). We tested the system using both randomly generated neural networks and networks with regular structures like unconstrained feed-forward ones, on various problems. Moreover, we also experimented with several non-differentiable error criteria.

The results show that solving the regularities problem using the constraints generated by our FT-CLP system considerably improves the convergence. In all the tests the automatic approach of separations based on \(<\) was used. We also tried \(<\), and the dynamic separations via \(<\), and \(<\). When refining the forward-tracking using these techniques, small improvements were achieved. We were able to design some particular constrained networks where these techniques lead to significant improvements: this, in our opinion, shows that (at least in the treatment of this problem) the simpler approach of separations based on \(<\) suffices for most of the cases, where the other more sophisticated techniques should be used if one wants to reach the highest performance in all the cases.

7 Conclusion

In this paper a novel search technique called forward-tracking, has been introduced, that allows to obtain solutions to programs whose original executions fail. Forward-tracking supports a modular approach of programming, where in order to formalize a problem, its various parts are specified separately, and the relations and priorities among these parts are described by means of forward-trackings. As a consequence, forward-tracking is applicable to a large class of problems, like planning, scheduling, decision support, negotiation: instances of these problems that are over-specified are admissible, and in such cases a satisfactory relaxation of the conflicting parts has to be provided. From a software engineering point of view, forward-tracking supports software reusability (cf. [11]). This was illustrated in the application we have considered, where programs developed to solve a particular case (one regularity, one layer) have been re-used with almost null modifications, obtaining a system able to cope with the general case of neural networks.

Acknowledgement The work of Elena Marchiori was partially supported by NWO, the Dutch Organization for Scientific Research, under grant 612-32-001.

References