INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig–
Maximilians–
Universität–
München–

A Confluence Test for Concurrent Constraint Programs

Slim Abdennadher, Thom Frühwirth, Michael Marte, Holger Meuss

http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1995-4, Oktober 1995

A Confluence Test for Concurrent Constraint Programs

Slim Abdennadher[1], Thom Frühwirth[2], Michael Marte[1], Holger Meuss[1]

[1]Ludwig-Maximilians-Universität, Munich, Germany
{Slim.Abdennadher,Michael.Marte,Holger.Meuss}@informatik.uni-muenchen.de

[2]ECRC, Arabellastrasse 17, Munich, Germany
Thom.Fruehwirth@ecrc.de

August 21, 1997

Abstract

We address the problem of identifying confluent parts of concurrent constraint programs. The confluence property guarantees that a concurrent program will always compute the same result independent of the execution strategy.

In this paper, we introduce a simple test for confluence based on work in rewrite systems. Furthermore, we show how to use this test to certify the combination of concurrent constraint programs where the same predicate is defined more than once.

1 Introduction

Concurrent constraint (CC) programming [8, 9] brings together ideas from constraint and concurrent logic programming [10]. One problem with CC languages is that it is difficult to reason about and to analyze concurrent programs [4]. Recent work, e.g. [6] (which also contains more references), has tried to solve the problem by giving a declarative semantics to CC programs. By "declarative semantics" we mean a semantics that admits a logical interpretation. The idea to make CC languages declarative is to introduce a non-standard semantics which is confluent. Basically, confluence guarantees that the outcome of a computation will be the same, independent of the execution strategy.

Our work complements this line of work by giving a simple test that is able to identify parts of a CC program which are confluent already in the standard semantics. In other words, the test allows to distinguish between the declarative and non-declarative parts of CC programs. By the way, very recent work in program analysis [2] starts from a subset of CCP that is confluent under a standard semantics. Our findings are the result of adapting work in conditional term rewrite systems (CTRS) [1, 5, 7] for CC languages. We will also detail the subtleties involved when transferring a theorem about confluence in CTRS to CC languages.

In addition it turns out that our confluence test is also useful when combining program parts (libraries or modules) in which the same predicate is defined (totally or partially) several times. A variant of the test can ensure that the different definitions are "compatible" and hence that the program parts can be combined by simply taking their union.

The paper is organized as follows. The next section gives the preliminaries of conditional rewriting and of concurrent constraint programming. Section 3 presents the notion of confluence for CC programs. Section 4 deals with implementation issues of the confluence test presented. Section 5 considers a practical application of the confluence test in combining systems. Section 6 concludes with a summary and directions for future work.

2 Preliminaries

2.1 Conditional Rewriting

We will give a quick introduction into the basic concepts of conditional term rewriting and confluence following [1]. For surveys on the topic consider [5, 7].

Definition. A *conditional (term) rewrite system* (CTRS) is a set of conditional rewrite rules. A *conditional rewrite rule* is a (first-order) formula of the form

$$u_1 \downarrow v_1 \wedge \ldots \wedge u_n \downarrow v_n \mid l \rightarrow r,$$

where $l, r, u_1, \ldots, u_n, v_1, \ldots, v_n$ are (first-order) terms built from a set F of function symbols and a set X of variables. The formula is (implicitly) universally quantified at the outermost scope. The subformulae $u_1 \downarrow v_1, \ldots, u_n \downarrow v_n (n \geq 0)$ are called *conditions*. We will abbreviate conditions by writing, instead, $u \downarrow v \mid l \rightarrow r$.

Definition. A *rewrite step* is performed by applying a rewrite rule to a term. A term t *rewrites* to a term t' by means of a rewrite rule $(u \downarrow v \mid l \rightarrow r)$ by replacing an instance $l\sigma$ of the left-hand side l in the term t by the corresponding instance $r\sigma$ of the right-hand side r, provided the corresponding instance $u\sigma \downarrow v\sigma$ of the condition holds, i.e. $u\sigma$ and $v\sigma$ rewrite to the same term w in zero or more steps.

Note that it is not possible to instantiate variables of the rewritten term t in a rewrite step, instead we only *match* a subterm of t against l.

We write $s \rightarrow t$, if s rewrites to t in one step; $s \rightarrow^* t$, if s rewrites to t in zero or more steps, we say t is *derivable* from s.

Two terms are said to be *joinable*, if they derive the same term, i.e. $s \rightarrow^* w$ and $t \rightarrow^* w$ for some term w. We write $s \downarrow t$, if s and t are joinable.

Example. [7] The following is a system for computing max and the relation $>$ on natural numbers. Natural numbers are represented as terms of the form $s^i(0)$, where s is the successor function.

$$\begin{array}{rcll}
(x > y) \downarrow true & \mid & max(x, y) \rightarrow x & (1) \\
(x > y) \downarrow false & \mid & max(x, y) \rightarrow y & (2) \\
true \downarrow true & \mid & (s(x) > s(y)) \rightarrow (x > y) & (3) \\
true \downarrow true & \mid & (s(x) > 0) \rightarrow true & (4) \\
true \downarrow true & \mid & (0 > x) \rightarrow false & (5)
\end{array}$$

The rewrite step for computing the maximum of $s(0)$ and $s(s(s(0)))$ is

$$max(s(0), s(s(s(0)))) \stackrel{(2)}{\rightarrow} s(s(s(0))).$$

Rule (1) is not applicable because its condition is not satisfied. Rules (3), (4) and (5) are not applicable because max does not match against $>$. Rule (2) is applicable because the atoms of the condition are joinable:

$$(s(0) > s(s(s(0)))) \stackrel{(3)}{\rightarrow} (0 > s(s(0))) \stackrel{(5)}{\rightarrow} false.$$

Definition. A rewrite system is said to be *terminating* (or *noetherian*) if there are no infinite derivations $t_1 \rightarrow t_2 \rightarrow \ldots$.

The above example is terminating, since - roughly speaking - the left-hand sides of the rewrite rules are always smaller in size than the right-hand sides and the conditions.

Definition. A rewrite system is said to be *confluent* (or *Church-Rosser*), if terms are joinable whenever they are derivable from the same term.

Definition. If the left-hand side g of a rule $(p' \downarrow q' \mid g \rightarrow d)$ unifies, via most general substitution σ, with a non-variable subterm s in a left-hand side l of a rule $(p \downarrow q \mid l \rightarrow r)$, then the conditional equation

$$(p\sigma, p'\sigma) \downarrow (q\sigma, q'\sigma) \mid l\sigma[d\sigma] = r\sigma$$

is called a *critical pair* of the two rules, where $l\sigma[d\sigma]$ is obtained by replacing s in l by d and applying σ.

Definition. A critical pair $(c \downarrow d \mid s = t)$ is said to be *joinable* if $s\sigma \downarrow t\sigma$ for any substitution σ such that $c\sigma \downarrow d\sigma$. A critical pair $(c \downarrow d \mid s = t)$ is said to be *feasible* if there is a substitution σ for which $c\sigma \downarrow d\sigma$.

Example. The rewrite system for *max* and $>$ has one critical pair. The heads of the rules (1) and (2) unify via the substitution $\sigma = id$ resulting in the critical pair

$$(x > y, x > y) \downarrow (true, false) \mid x = y.$$

This is a so called *trivial* critical pair for it is not feasible (under the usual interpretation of $>$ as strict order relation).

Example. [1] The following is a system for computing the relation $\leq$ on natural numbers.

$$\begin{array}{rcll} true \downarrow true & \mid & 0 \leq 0 \rightarrow true & (6) \\ true \downarrow true & \mid & s(x) \leq 0 \rightarrow false & (7) \\ true \downarrow true & \mid & s(x) \leq s(y) \rightarrow x \leq y & (8) \\ u \leq v \downarrow true & \mid & u \leq s(v) \rightarrow true & (9) \end{array}$$

The heads of the rules (8) and (9) unify via the substitution $\sigma = \{u \rightarrow s(x), v \rightarrow y\}$, so we have the critical pair

$$(true, s(x) \leq y) \downarrow (true, true) \mid x \leq y = true.$$

This critical pair is feasible, i. e. if the condition of rule (9) holds we can go the two ways

$$s(x) \leq s(y) \stackrel{(9)}{\rightarrow} true$$

and

$$s(x) \leq s(y) \stackrel{(8)}{\rightarrow} x \leq y \rightarrow^* true \text{ because } s(x) \leq y \downarrow true.$$

Hence this critical pair is joinable.

Definition. A critical pair is an *overlay* if it is obtained from two left-hand sides that unify at their roots.

In the above examples, we dealt with overlays only.

2.2 Concurrent Constraint Programming

We assume some basic familiarity with concurrent (constraint) programming (CCP) [8, 9, 10]. There is a distinguished class of predicates, the (built-in) *constraints*. There is a built-in constraint solver that solves, checks and simplifies these constraints.

Definition. A *CC program* is a finite set of CC clauses. A *CC clause* is an expression of the form

$$H \Leftrightarrow G_1, \ldots, G_m \mid B_1, \ldots, B_n$$ [1]

where the head H is an atom but not a constraint, and the guard $G_1, \ldots G_m$ is a conjunction of constraints and the body $B_1, \ldots, B_n$ is a conjunction of atoms called goals.

The *operational semantics* of CC can be described by a transition system.

Definition. A *computation state* is a tuple

$$< Gs, C_B >,$$

where Gs is the goal store (resolvent) and C_B is the constraint store for the constraints respectively. A store is a conjunction of atoms represented as a set.

The *initial state* consists of a query Gs and an empty constraint store,

$$< Gs, \{\} >.$$

A *final state* is either *failed* (due to an inconsistent constraint store represented by the unsatisfiable constraint `false`),

$$< Gs, \{\texttt{false}\} > \quad \text{(equivalent to } < \{\}, \{\texttt{false}\} >\text{)},$$

or *successful* if C_B is not `false`

$$< Gs, C_B >.$$

The following *computation steps* are possible to get from one computation state to the next.

Solve $< \{C\} \cup Gs, C_B > \ \longmapsto \ < Gs, C'_B >$
if $(C \wedge C_B) \leftrightarrow C'_B$

The constraint solver updates the constraint store C_B if a new constraint C was found in the goal store. To *update* the constraint store means to produce a new constraint store C'_B that is logically equivalent to the conjunction of the new constraint and the old constraint store. Given a CC program P.

Unfold $< \{H'\} \cup Gs, C_B > \ \longmapsto \ < Gs \cup B\sigma, C_B >$
if $(H \Leftrightarrow G \mid B)$ is a variant of a clause in P and there is a substitution σ with $H\sigma = H'$ and $C_B \rightarrow G\sigma$

[1] We use the syntax of [10] here.

To *unfold* an atomic goal H' means to add $B\sigma$ to Gs, provided H' matches the head of a CC clause $(H \Leftrightarrow G \mid B)$ via substitution σ and $G\sigma$ is satisfied under the constraint store C_B.

A guard G is satisfied $(C_B \rightarrow G)$ if it is logically entailed (implied) by the constraint store C_B. Entailment can be computed by testing if adding G leaves the constraint store unchanged, i.e. $< G, C_B > = < \{\}, C_B >$.

3 Confluence of CC Programs

Definition. A set of CC clauses is *confluent*, if each possible order of computation steps starting from an arbitrary computation state leads to the same final state.

In the following we assume that the built-in constraint solver is confluent, terminating and logically correct. This allows us to treat the solver as a black box.

Now we want to prove confluence of CC clauses by applying the following theorem in [1] to the CC formalism.

Theorem (Confluence of CTRS). *A terminating conditional rewrite system is confluent, if all its critical pairs are joinable overlays.*

To make the theorem applicable, we relate CC programs to rewrite systems: The role of terms is taken by computation states. A rewrite step has its equivalent in a computation step. Thus we can define:

Definition. A set of CC clauses is *terminating* if any possible application of *solve* and *unfold* results in a final state.

We define critical pairs for CCP analogously to critical pairs in CTRS, where CC guards take the role of CTRS conditions:

Definition. If the head H_1 of a CC clause $(H_1 \Leftrightarrow G_1 \mid B_1)$ unifies, via most general substitution σ, with the head H_2 of a another CC clause $(H_2 \Leftrightarrow G_2 \mid B_2)$ then the conditional equation

$$G_1\sigma \wedge G_2\sigma \mid B_1\sigma =?= B_2\sigma$$

is called a *critical pair* of the two CC clauses.

In CC programs every critical pair is obviously an overlay, because goal atoms can only match against heads of CC clauses and predicates do not occur inside atoms.

Definition. A critical pair $(G \mid B_1 =?= B_2)$ is said to be *joinable* if the computations of $< G \wedge B_1, \{\} >$ and $< G \wedge B_2, \{\} >$ result in the same final state. Like in a CTRS, if the computation is nondeterministic because different computation steps can be taken (by applying different clauses) from a given state, it suffices to consider one sequence of computation steps only.

Definition. A critical pair $(G \mid B_1 =?= B_2)$ is *feasible* if $< G, \{\} >$ does not fail.

It is easy to see that we now have an appropriate translation. If we apply this translation to the proof given in [1] we obtain the following result.

Theorem (Confluence of CC Programs). *A terminating concurrent constraint program is confluent, if all its critical pairs are joinable.*

It is important to note that we rely on the following facts, some of which we already mentioned:

Built-in constraints The built-in constraint solver with its termination, confluence and correctness properties guarantees that feasibility and joinability are detected. If one studies the `range` example in section 6, it is easy to see that these properties of the underlying constraint system are essential. If the implementation of `max` would not reduce the goals `Max<Min, max(Min,Max,Max)` to the normal form `false`, the joinability of critical pairs could not be detected.

Guard checking The guard G of a CC clause $(H \Leftrightarrow G \mid B)$ is an abbreviation for writing $< G, C_B > \downarrow < \texttt{true}, C_B >$. So guard checking in CC computation steps consists in checking joinability (like in CTRS).

ACI Goals and constraints in the stores are connected by conjunction, $\wedge$, which is associative, commutative and idempotent (ACI). However, these properties would only influence the application of the confluence theorem if it changed the computation steps that can be taken. In the operational semantics of CC programs, the ACI properties are expressed through the set notion. The AC properties enable us to choose any atom in a conjunction as the one to apply the next computation step (solve or unfold). This corresponds with the situation in a rewrite system, where any subterm can be rewritten, independent of its position. The restriction to overlays in the theorem corresponds with the fact that we can only unfold atoms with a program clause and not arbitrary terms appearing in arbitrary positions. Idempotence is also not a problem, since multiple occurrences of the same atom are trivially confluent if the associated predicate is confluent.

Join The joinability check of CC clauses compares final states, whereas in CTRS any states can be compared. The first condition seems stronger, but this is not the case: If we have two identical non-final states, according to the definition of joinability, we can always choose a sequence of computation steps for both of them such that they lead to the same final state. We can not "miss" any identical states by restricting ourselves to final states.

Local and global knowledge The checking of the guard in CTRS only requires local knowledge, namely knowledge of the identity of the subterm being dealt with. In CC clauses we have a different situation: The entailment test requires knowledge of the current state of the constraint store, i.e. global knowledge. This means (translated to CTRS terminology) that knowledge not only of the subterm but also of the context of the subterm is required. However, the proof of the theorem does not rely on the fact that only local knowledge is available.

Matching In CTRS we can rewrite terms containing variables. But it is important to note that these variables will never be instantiated by application of rewrite rules. This meets with the restriction in the computation step *unfold*, where matching of the goal atom with the respective head is required. Of course, variables can be further constrained and instantiated in the body of a CC clause.

4 Implementation Issues

Usually, not all parts of a CC program will be confluent. We therefore restrict ourselves to confluent predicates.

Definition. A CC predicate p is *weakly confluent*, if all critical pairs of CC clauses with p in the head are joinable.

It is clear that a CC program is confluent if all its predicates are weakly confluent. However, if only some predicates are confluent, we can only speak of weak confluence, since the confluent predicates may be defined in terms of some non-confluent predicates.

Definition. A CC predicate p is *strongly confluent*, if it is weakly confluent and all predicates occurring in the bodies (except p itself) are strongly confluent.

Strong confluence of a predicate it is not a local property anymore, since it depends on the program as a whole, i.e. on the confluence of other predicates. For similar definitions of confluence see also the recent [2].

We are currently developing a tool in ECL^{i}PS^e(ECRC Constraint Logic Programming Platform) which tests confluence of constraint handling rules (CHRs) [3]. CC programs are a subset of the CHR language. Th implementation has its theoretical foundation in the theorem given above, and checks joinability of critical pairs of a given predicate.

In the following we will present the basic structure of our program. First, the CC clauses of the program are both loaded and compiled as well as made explicit as facts `rule/4`. The test for weak confluence of a predicate `pred` is started with the query `confluence(pred)`.

`confluence(Pred)` tests if `Pred` is weakly confluent and outputs critical pairs that are not joinable. The potential critical pairs are generated by taking two clauses of the CC program and unifying their heads (`Head1=Head2`).

`feasible(Guard1,Guard2,Guard)` succeeds if the conjunction of `Guard1` and `Guard2` simplifies to `Guard`. To compute `Guard`, we use a predicate `localcall(Goal,F)` that locally computes and returns the final state (the goal and constraint store) `F` of a query `Goal`. If there is more than one final state, they are returned on backtracking. `localcall/2` does not instantiate variables in `Goal` but rather represents substitutions as equalities appearing in `F`. It is implemented using the built-in predicate `subcall/2` of ECL^{i}PS^e. If a critical pair is feasible it is tested for joinability.

`joinable(Head,Body1,Body2,Guard)` is true if `Body1` and `Body2` are joinable under the condition that the guard `Guard` is true. Again, `localcall/2` is used to locally execute each of the bodies together with the combined guard. Since the CC program has been loaded as well, we can execute the bodies directly without interpretation overhead. Then the result of both computations is compared. It is checked that the results are variants, i.e the same up to renaming of variables. If the pair is not joinable it is displayed together with the rules it derives from.

```
confluence(Pred):-

    rule(Pred,Head1,Guard1,Body1),      % take two CC clauses
    rule(Pred,Head2,Guard2,Body2),

    Head1=Head2,                        % critical pair possible?

    feasible(Guard1,Guard2,Guard),

    not joinable(Guard,Body1,Body2),

    % pretty print output of critical pair that is not joinable

    false.                              % test all clause pairs

confluence(_Pred).                      % done


feasible(Guard1,Guard2,Guard):-

    localcall((Guard1,Guard2),Guard).
```

```
joinable(Guard,Body1,Body2):-

    localcall((Guard,Body1),Result1),
    localcall((Guard,Body2),Result2),

    variant(Result1,Result2).        % Results are the same?
```

Example

The following example is an implementation of merge, i.e. merging two lists into one list as the elements of the input lists arrive. Thus the order of elements in the final list can differ from computation to computation.

```
merge([],L2,L3) <=> true | L2=L3.
merge(L1,[],L3) <=> true | L1=L3.
merge([X|L1],L2,L3) <=> true | L3=[X|L],merge(L1,L2,L).
merge(L1,[X|L2],L3) <=> true | L3=[X|L],merge(L1,L2,L).
```

There are 4 critical pairs. For example, the critical pair coming from the first two clauses is

`true | (L3=[]) =?= (L3=[])`

stemming from any query which is an instance of `merge([],[],L3)`. Obviously this critical pair is joinable.

If `merge/3` meets the specification, there is also space for nondeterminism that causes non-confluence. Indeed, our confluence tester produces one critical pair that is not joinable:

```
:-confluence(merge).

CRITICAL PAIR, NOT JOINABLE:

true |
(L3 = [X|L], merge(L1, [Y|L2], L))
=?=
(L3 = [Y|L], merge([X|L1], L2, L))

Differing final states:

(L3 = [X|L], L = [Y|L'], merge(L1, L2, L'))
=?=
(L3 = [Y|L], L = [X|L'], merge(L1, L2, L'))

Used clauses:

* merge([X|L1], [Y|L2], L3) <=> true | L3 = [X|L], merge(L1, [Y|L2], L)
instance of
merge([X|L1], L2, L3) <=> true | L3 = [X|L], merge(L1, L2, L)

* merge([X|L1], [Y|L2], L3) <=> true | L3 = [Y|L], merge([X|L1], L2, L)
instance of
merge(L1, [X|L2], L3) <=> true | L3 = [X|L], merge(L1, L2, L)
```

```
No (more) unjoinable critical pairs.
yes.
```

It can be seen from the output of our confluence tester that a query like `merge([X|L1], [Y|L2], L3)` can either result in putting `X` before `Y` in the output list `L3` or vice versa, hence - not surprisingly - `merge/3` is not confluent.

5 Combining Concurrent Constraint Programs

We want to combine CC programs which overlap in the definition of some predicate p. A typical scenario is that of modules or libraries implementing similar functionality. Clearly, we want to make sure that the operational semantics of both definition of p do not differ. We can use our confluence test to ensure that.

Definition. Two sets of CC clauses defining the same predicate p are *compatible* if all the critical pairs coming from taking one clause of each set are joinable.

If the confluence test fails, we can locate the clauses responsible for the problem. If the test succeeds, we can just take the union of the clauses in both programs. This means that the predicate p can even be partially defined in the programs which are combined.

Example

The following example is a combination of two CC programs S_1 and S_2 overlapping in the definition of `range(X,Min,Max)`, which is true if the value of `X` is between `Min` and `Max`.

S_1 contains the following CC clause defining `range`:

```
range(X,Min,Max) <=> true |
                     max(Min,Max,Max), max(X,Min,X), max(X,Max,Max).
```

whereas S_2 has the following definition of `range`:

```
range(X,Min,Max) <=> Max<Min | false.
range(X,Min,Max) <=> Min=<Max | Min=<X, X=<Max.
```

We want to know whether the definitions of `range` are compatible. We assume that the underlying constraint solver defines the constraints `<`, `=<`, `max` , `=` and that it is terminating, confluent and logically correct. There are two critical pairs, coming from taking the only clause for `range` in S_1 with one of the clauses in S_2. Since these critical pairs are joinable, the two definitions of `range` are compatible and hence we can just take the union of the clauses and define `range` by all three clauses[2].

6 Conclusions

We think that our simple confluence test can identify confluent and hence declarative parts of concurrent (constraint) programs. This information should make it easier to reason about and to analyze concurrent programs [4]. Our approach also nicely complements recent work that gives confluent, non-standard semantics for CC languages to make them amenable to abstract interpretation and analysis in general, since our confluence test can find out parts of the program which are confluent already under the standard semantics.

[2]Note that the confluence test ensures compatibility, it does not deal with the undecidable subsumption of clauses and predicates.

Furthermore, as we have illustrated, confluence is also interesting from the software engineering point of view. A variant of the confluence test can guarantee that the combination of program parts (like modules or libraries) is just the union of the clauses, even if some confluent predicates are fully or partially defined in several program parts.

With the work presented, we started to investigate the confluence of constraint handling rules (CHRs) [3], a superset of CC languages designed for writing constraint solvers. (Interestingly, in [4] it is remarked that CC atoms in general are best seen as constraints). For this purpose, CHRs allow for "multiple heads", i.e. conjunctions of atoms in the head, and for propagation rules, i.e. rules that do not rewrite atoms, but only add additional atoms. Since we are only concerned with writing constraints, all CHR clauses have a declarative reading as well. Confluence and logical correctness of such user-written constraints seem to be closely related. Moreover, when combining constraint solvers, one is often confronted with constraints that are partially defined in several solvers.

Acknowledgements. Thanks to Hélène Kirchner for detailed comments on a rough draft on confluence of CHRs that lead to the work described here. Thanks to Esther Brichta for the rapid fax service.

References

[1] N. Dershowitz, N. Okada, and G. Sivakumar. Confluence of conditional rewrite systems. In *1st CTRS*, pages 31–44. LNCS 308, 1988.

[2] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence in concurrent constraint programming. In Alagar and Nivat, editors, *Proceedings of AMAST '95, LNCS 936*. Springer, 1995.

[3] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*. LNCS 910, March 1995.

[4] C. Hewitt and G. Agha. Guarded horn clause languages: Are they deductive and logical. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 650–657, Ohmsha, Tokyo, 1988.

[5] Claude Kirchner and Hélène Kirchner. *Rewriting: Theory and Applications*. North-Holland, 1991.

[6] K. Marriott and M. Odersky. A confluent calculus for concurrent constraint programming with guarded choice. In Ugo Montanari Francesca Rossi, editor, *Principles and Practice of Constraint Programming, Proceedings First International Conference, CP'95, Cassis, France*, pages 310–327, Berlin, September 1995. Springer.

[7] David A. Plaisted. Equational reasoning and term rewriting systems. In D. Gabbay, C. Hogger, J. A. Robinson, and J. Siekmann, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1, chapter 5, pages 273–364. Oxford University Press, Oxford, 1993.

[8] V. A. Saraswat, M. Rinard, and P. Panangaden. The semantics foundations of concurrent constraint programming. In *Conference Record of the Eighteenth Annual ACM Symposium on principles of Programming Languages*, pages 333–352, Orlando, Florida, January 1991. ACM Press.

[9] V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, Cambridge, 1993.

[10] E. Shapiro. The family of concurrent logic programming languages. In *ACM Computing Surveys*, volume 21:3, pages 413–510, September 1989.