Appeared in the Proceedings of the Post-conference workshop at ICLP: Verification and analaysis of (concurrent) logic languages, June 1994, eds. F.S. de Boer and M. Gabbrielli, pp. 87–107

Towards the Verification of Concurrent Constraint Programs in Finite Domain Reasoning

Jörg Würtz

German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
D-66123 Saarbrücken
Germany
E-mail: wuertz@dfki.uni-sb.de

Abstract

Combinatorial problems occur quite often in many application areas, e.g. scheduling or cutting stock problems, and may be specified as problems over finite domains, i.e., the variables range over a finite subset of the integers. Our approach taken in the concurrent constraint language Oz is to support an efficiently implementable constraint system for constraints $x \dot{\in} D$. Using this constraint system, the programmer can add his own functionality by so called virtual constraints (e.g. $x \leq y$). Because the problem of satisfiability of these more complex constraints is NP-complete, most of them are implemented incompletely by using reflecting operators returning the current minimum and maximum of a variable's domain.

For these programs serving as the core of finite domain reasoning, the question of verification arises. Verification is complicated by the operators reflecting the current known information about variables. In our approach we translate the computation states into first order formulas by giving virtual constraints their intended semantics (the semantics of the corresponding logic constraint). We prove that for a reduction of computation states $\gamma \rightarrow \gamma'$ the corresponding first order formulas are equivalent with respect to the intended model. Our approach, which seems to be very natural in the setting of constraint languages, succeeds in proving virtual constraints to be correct, terminating and complete (in the shown example) with respect to an intended semantics.

1 Introduction

Combinatorial problems occur in many application areas, e.g. scheduling or cutting stock problems. They may be specified as problems over finite domains, i.e., the variables range over a finite subset of the integers. On one hand, there are basic constraints $x \dot{\in} D$, where D is a finite subset of the integers. For basic constraints there are efficient and incremental algorithms for deciding satisfiability and entailment. But on the other hand, one wants to have more complex constraints like $x \leq y$ or $x + y = z$, which we call virtual constraints. Because of the complexity of solving these constraints (NP-complete), they are usually implemented by incomplete algorithms.

The language considered in this paper is a subset of Oz [HSW93, SHW93, Smo94, HMM$^+$94]. Oz is an attempt to create a high-level concurrent programming language providing the problem solving capabilities of logic programming. We have implemented virtual constraints by Oz programs

[SSW94]. Hence, we have the possibility to analyze and test our algorithms on a high level before implementing them (e.g. as builtins). A variety of algorithms may be designed and tested with small effort.

Since algorithms for virtual constraints are at the core of languages like CHIP [DVS+88] or `cc(FD)` [VSD93] the question arises whether these algorithms are correct, terminating and complete.[1] In the sequential setting general algorithms achieving arc-consistency (e.g. [Mac77, VDT92]) are proved to be correct. Nevertheless, the concurrent logic setting is the more natural one for solving constraint problems because of the possibility to deal with incomplete information in a reactive system. Whereas much work is done in the field of verification in the setting of imperative concurrent languages (e.g. [MP92]), this work has just begun in the field of concurrent logic languages (see for example [dBGMP94]).

The implementation of virtual constraints in Oz is based on a constraint solver for basic constraints $x \dot{\in} D$, the suspension mechanism of the language (ask-clauses in the CC framework [SR90]) and reflecting operators (to the sake of efficiency). The reflecting operators reflect the actual constraints for a variable, e.g. the actual minimal value a finite domain variable can take consistently. Our approach towards verification of concurrent constraint programs comes naturally by observing that virtual constraints should have a declarative specification. We translate computation states (consisting of (virtual) constraints, procedure definitions, applications, conditionals etc.) into a first order formula (observe that in this paper we only consider a subset of Oz by excluding higher-order programming, deep guards and state). To this aim we define a model, which is a persistent extension of the structure of the basic constraint system, by giving the procedures occurring in the definition of a virtual constraint a declarative semantics (and also the reflecting operators). One can prove that for a reduction of computation states $\gamma \rightarrow \gamma'$ the obtained first order formulas of γ and γ' are equivalent with respect to this model. Beside proving correctness, one can prove termination and in special cases completeness of the considered virtual constraint, i.e., if the computation does not fail, the formula obtained from the inital state is satisfiable.

The paper is structured as follows. In Section 2 we outline the calculus underlying the considered subset of Oz, show the virtual constraints, which are subject of this paper, and sketch the used proof techniques. The calculus is formally described in Section 3. In Section 4 the proof techniques are elaborated further. The paper concludes with an outlook.

2 Outline

In this section we learn in an informal way the constraint system $\mathcal{FD}$, explain the underlying computation model of the considered subset of Oz and see a so called virtual constraint, subject of this paper. For the sake of clarity, we chose as a virtual constraint a simple one, which can be proved to be complete. At the end of this section we sketch the techniques to prove the virtual constraint to be correct, terminating and complete.

2.1 The Constraint System $\mathcal{FD}$

Let *Inf* and *Sup* be integers with $Inf \leq Sup$. The constraint system $\mathcal{FD}$ consists of the signature $\Sigma = \{\dot{\in} D | D \subseteq \{Inf, \ldots, Sup\}\}$ of unary predicate symbols and the structure $\mathcal{A}$ over Σ, where the universe of $\mathcal{A}$ consists of the integers $\mathbb{Z}$ and $x \in \dot{\in}^{\mathcal{A}} D$ iff x is an element of D. The equality symbol $\doteq$ is a binary predicate that is always interpreted as identity. A basic

[1]Languages like `clp(FD)` [DC93] using the concept of indexicals do not use virtual constraints in our sense but the question for verification also arises there.

constraint is defined as follows:

$$\phi, \psi ::= \top \mid \bot \mid x \doteq y \mid x \dot{\in} D \mid \exists x \phi \mid \phi \wedge \psi.$$

It is possible to transform a constraint $\exists \tilde{x} \phi$ into a normal form:

$$\psi = x_1 \doteq y_1 \wedge \ldots \wedge x_n \doteq y_n \wedge x_{n+1} \dot{\in} D_1 \wedge \ldots \wedge x_{n+m} \dot{\in} D_m$$

such that $x_i, 1 \leq i \leq n+m$, occurs exactly once or $\psi = \bot$. In case a constraint ψ is in such a normal form, D_i is called the domain of x_{n+i}. We will say a variable x is determined in case the normal form of a constraint contains $x \dot{\in} D$ with $|D| = 1$.

We now want to consider some examples. Obviously, $x \dot{\in} \{1,2\} \wedge x \dot{\in} \{2,3\}$ is satisfiable (with $x = 2$ as sole solution), and $x \dot{\in} \{1,2\} \wedge x \dot{\in} \{4,7,9\}$ is unsatisfiable in $\mathcal{A}$. Obviously, $x \dot{\in} \{2,3\} \models_{\mathcal{A}} x \dot{\in} \{1,2,3\}$ where $\phi \models_{\mathcal{A}} \psi$ if the structure $\mathcal{A}$ satisfies $\tilde{\forall}(\phi \rightarrow \psi)$, since each solution of $x \dot{\in} \{2,3\}$ is a solution of $x \dot{\in} \{1,2,3\}$. On the other hand, $x \dot{\in} \{1,2,3\} \models_{\mathcal{A}} \neg(x \dot{\in} \{8\})$ since no solution of the left-hand side is a solution of the right-hand side and $x \dot{\in} \{1,2,3\} \wedge y \dot{\in} \{1,2,3\} \not\models_{\mathcal{A}} x \doteq y$ since e.g. the valuation $\{x \mapsto 1, y \mapsto 2\}$ is a solution of the left-hand side but not of the right-hand side.

2.2 The Computation Model

In [Smo94] a formal model of computation in Oz is given, consisting of a calculus rewriting expressions modulo a structural congruence relation, similar to the setup of the π-calculus [Mil91]. The usual distinction between program and query is alleviated. For the purposes of this paper, we simplify the calculus by not considering deep guards, higher-order programming and state. The following is adopted from [SSW94].

A computation space consists of a number of actors connected to a blackboard. The actors read the blackboard and reduce once the blackboard contains sufficient information. The information on the blackboard increases monotonically. When an actor reduces, it may put new information on the blackboard and create new actors. As long as an actor does not reduce, it does not have an outside effect. The actors of a computation space are short-lived: once they reduce they dissapear.

The blackboard stores a constraint and a number of abstractions. The constraint on the blackboard is always satisfiable. We say that a blackboard entails a constraint ψ if the implication $\phi \rightarrow \psi$ is valid, where ϕ is the constraint stored on the blackboard. We say a blackboard is consistent with a constraint ψ if the conjunction $\phi \wedge \psi$ is satisfiable with respect to a given model.

There are several kinds of actors. An elaborator is an actor executing an expression. Elaboration of a constraint ϕ checks whether ϕ is consistent with the blackboard. If this is the case, ϕ is conjoined to the constraint on the blackboard; otherwise,the computation space is marked failed and all its actors are cancelled. Elaboration of a constraint corresponds to the eventual tell operation of CC [SR90].

Elaboration of a concurrent composition $\sigma_1\ \sigma_2$ creates two separate elaborators for σ_1 and σ_2.

Elaboration of a variable declaration **`local`** x **`in`** σ **`end`** creates a new variable and an elaborator for the expression σ. Within the expression σ the new variable is referred to by x. Every computation space maintains a finite set of local variables.

Elaboration of a procedure definition **`proc`** $\{p\ x_1\ \ldots\ x_n\}\ \sigma$ **`end`** writes the abstraction $p : y_1 \ldots y_n / \sigma$ on the blackboard.

Elaboration of a procedure application $\{p\ y_1\ \ldots\ y_n\}$ waits until the blackboard contains the abstraction $p : x_1 \ldots x_n / \sigma$. When this is the case, an elaborator for the expression $\sigma[y_1/x_1 \ldots y_n/x_n]$

is created ($\sigma[y_1/x_1 \dots y_n/x_n]$ is obtained from σ by replacing the formal arguments $x_1 \dots x_n$ with the actual arguments $y_1 \dots y_n$).

Elaboration of the expression $\{\min\ x\ y\}$ ($\{\max\ x\ y\}$) reduces to an actor waiting for information on the blackboard on the lower (upper) bound of x (y). If there is information on the blackboard, this actor reduces to an elaborator for $y \doteq n$ ($y \doteq m$), where $n = \max(k | \phi \models_{\mathcal{A}} x \dot{\in} \{k, \dots, Sup\})$ ($m = \min(k | \phi \models_{\mathcal{A}} x \dot{\in} \{Inf, \dots, k\})$), i.e., the greatest lower bound of x (smallest upper bound of y) with respect to the actual constraint ϕ on the blackboard.

Elaboration of a (non-deterministic) conditional expression **if** δ_1 [] ... []δ_n **else** σ **fi** creates a conditional actor. A clause δ_i takes the form $x_1 \dots x_k$ **in** ϕ **then** σ where the local variables $x_1 \dots x_k$ range over both the guard ϕ and the body σ of the clause. The conditional actor waits until the blackboard entails either a ϕ_i or $\neg\phi_i$. In the first case the actor reduces to an elaborator for **local** $x_1 \dots x_k$ **in** ϕ σ **end**. In the second case, the clause is simply discarded. If all clauses of a conditional actor **if** δ_1 [] ... []δ_n **else** σ **fi** are discarded, the conditional reduces to an elaborator for the expression σ .

2.3 Virtual Constraints

So far we have seen basic constraints like $x \dot{\in} D$. For basic constraints there are efficient and incremental algorithms for deciding satisfiability and entailment. Their semantics is given purely declaratively. One one hand, one wants to have for solving finite domain problems constraints like $x \leq y$ or $x + y = z$. On the other hand, it is well known that satisfiability of conjunctions of addition and multiplication of integers out of a finite domain is very costly (NP-complete). The usual way to deal with this problem is to base the implementation of more complex constraints on incomplete algorithms. They are not fully characterized by their declarative semantics and the programmer must know their operational semantics. We call such constraints virtual constraints and they are implemented as programs whose operational semantics is sound but incomplete with respect to the declarative semantics of the corresponding logic constraint.

As an example of a virtual constraint we consider $x \leq' y$ with the declarative specification $x \leq y$, which is relatively simple to implement (by a complete algorithm). Along this example we explain the language and sketch our proof techniques. Program 2.1 is an implementation of $x \leq' y$.

Due to the specification of $x \leq' y$ the computation space should be failed for $x \leq' y \wedge x \dot{\in} \{4, 5\} \wedge y \dot{\in} \{1, 2, 3\}$ due to the unsatisfiability of its declarative reading. For $x \leq' y \wedge x \dot{\in} \{2, 3, 4\} \wedge y \dot{\in} \{1, 2, 3\}$, the computation space must not be failed because of the satisfiability of the conjunction's declarative reading.

We now want to get an idea what Program 2.1 is doing. In the procedure $\leq'$, two variables are declared, which are bound to the current minimum of the domain of X and maximum of the domain of Y , respectively. *greater*, *gec*, *less*, *lec* denote constraints, which are elaborated if their second argument is constrained to an integer (observe that they denote unary constraints – one for each integer); we have chosen the syntax for procedure application only for convenience. $\{gec\ X\ V\}$ denotes the constraint $X \dot{\in} \{V, \dots, Sup\}$, $\{greater\ X\ V\}$ denotes $X \dot{\in} \{V{+}1, \dots, Sup\}$ and $\{lec\ X\ V\}$ denotes $X \dot{\in} \{Inf, \dots, V\}$ etc. Thus, the lower bound of the domain of X becomes the (new) lower bound of Y (the upper bound of Y becomes the (new) upper bound of X). The intuition behind the procedures *propLow* and *propUp* is to watch the lower bound of X and the upper bound of Y , respectively. E.g. the conditional in the procedure *propLow* waits until the lower bound of X is raised, i.e., is greater than the previously computed bound. If this is the case, a new local variable U is declared and bound to the current minimum of X's domain. $\{gec\ Y\ U\}$ will raise also the lower bound of Y . Then, the procedure is called recursively with the new lower bound for X . In case $X \dot{\in} \{V + 1, \dots, Sup\}$ is disentailed, the procedure reduces

Oz-Program 2.1 $X \leq' Y$

```
proc {=<' X Y}
   local XMin YMax in
      {min X XMin} {max Y YMax}
      {gec Y XMin} {lec X YMax}
      {propLow X Y XMin}
      {propUp X Y YMax}
   end
end

proc {propLow X Y XMin}
   if {greater X XMin}
   then local U in
           {min X U}
           {gec Y U}
           {propLow X Y U}
        end
   else true
   fi
end
proc {propUp X Y YMax}
   if {less Y YMax}
   then local U in
           {max Y U}
           {lec X U}
           {propUp X Y U}
        end
   else true
   fi
end
```

to **true**. Similar in procedure *propUp* the upper bound of Y is observed.

As a concrete example consider

$$\{\leq' \ X\ Y\}\ \{gec\ X\ 1\}\ \{lec\ X\ 5\}\ \{gec\ Y\ 2\}\ \{lec\ Y\ 4\}.$$

Elaboration of this expression will result in a blackboard such that the basic constraints are equivalent to $X\dot{\in}\{1,\ldots,4\}\wedge Y\dot{\in}\{2,\ldots,4\}$. Elaboration of an additional constraint $X\dot{\in}\{3,4\}$ leads to entailment of the conditional's clause defined in the procedure *propLow* such that $Y\dot{\in}\{3,\ldots,Sup\}$ will be elaborated leading to $Y\dot{\in}\{3,4\}$.

2.4 Sketch of Proof Techniques

The key idea of our verification technique is to translate the computation state γ consisting of the blackboard, actors and the not yet elaborated expressions into a first-order formula. As said in the previous section, the computation calculus consists of a set of reduction rules modulo a structural congruence (e.g. logical equivalence for constraints or α-renaming). We say that a transformation of computation states by congruence or reduction rules is an equivalence transformation if the corresponding first order formulas are equivalent with respect to a fixed model. We can prove that in the context of the considered virtual constraints the congruence and reduction rules are equivalence transformations with respect to an intended model (e.g. the virtual constraint $x \leq' y$ should have the semantics $x \leq y$). Note that the reduction rules are not equivalence preserving in general because of the reflective operators min and max[2] and possible non-deterministic conditionals. The reason for using an intended model is that virtual constraints should have a concise declarative semantics and dealing with reflective operators becomes easier. We prove that the considered program obeys the declarative specification (correctness), the program terminates, and the computation space fails if the corresponding first order formula is unsatisfiable (completeness).

We first define a persistent extension $\mathcal{B}$ of $\mathcal{A}$ (cf. Section 2.1), extended by the relations used for the virtual constraint (e.g. $\{\min\ X\ Y\}$ gets the semantics $Y \leq^{\mathcal{A}} X$ and $\{propLow\ X\ Y\ Z\}$ the semantics $X \leq^{\mathcal{A}} Y \wedge Z \leq^{\mathcal{A}} X$). To prove the correctness we show that every reduction rule is an equivalence transformation with respect to the intended model $\mathcal{B}$, if the computation is started in a state not containing min / max-operators outside of abstractions. To this aim we translate a computation state γ into a first-order formula. A mapping $[\![\,]\!]$ from computation states to formulas is defined such that

$$[\![\mathbf{proc}\ \{p\, x_1 \ldots x_n\}\ \sigma\ \mathbf{end}]\!] := \forall x_1 \ldots x_n (p(x_1 \ldots x_n) \leftrightarrow [\![\sigma]\!])$$

$$[\![\{p\, x_1 \ldots x_n\}]\!] := p(x_1 \ldots x_n)$$

$$[\![\mathbf{if}\ x_1 \ldots x_n\ \mathbf{in}\ \phi\ \mathbf{then}\ \sigma\ \mathbf{else}\ \tau\ \mathbf{fi}]\!] := \exists x_1 \ldots x_n (\phi \wedge [\![\sigma]\!])\ \vee\ (\neg \exists x_1 \ldots x_n \phi) \wedge [\![\tau]\!]$$

for procedure definitions, procedure calls and deterministic conditionals. We first prove that for the congruence $\equiv$ it holds that $\gamma \equiv \gamma' \Rightarrow [\![\gamma]\!] \models\!\mid_{\mathcal{B}} [\![\gamma']\!]$.

We then show that procedure application (also called unfolding) is an equivalence transformation. Next we prove that the used reflective rules are equivalence transformations in the context of the considered virtual constraint. Since it is obvious that the other reduction rules are equivalence preserving, it follows that $\gamma \rightarrow \gamma' \ \Rightarrow\ [\![\gamma]\!] \models\!\mid_{\mathcal{B}} [\![\gamma']\!]$ for all reduction steps with respect to the considered virtual constraint $\leq'$. This proves the correctness. Graphically, this can be stated as shown in Figure 1.

[2] $\{\min\ X\ Y\}$ results in $Y \doteq 5$ if the current blackboard entails $X\dot{\in}\{5,6,7\}$ and results in $Y \doteq 6$ if the blackboard entails $X\dot{\in}\{6,7\}$ later on.

$$\begin{tikzcd}
\gamma \arrow[r] \arrow[d] & \gamma' \arrow[d] \\
[\![\gamma]\!] \arrow[r, phantom, "\models\!\!\!\dashv_{\mathcal{B}}"] & [\![\gamma']\!]
\end{tikzcd}$$

Figure 1: Principle of correctness proof

Moreover, one can prove that the considered virtual constraint terminates (essentially by the finiteness of the domains and the right operational behavior of the reflective operators).

We call a program complete if in case the computation space is not failed during reduction, the corresponding first order formula of the initial state is satisfiable in the model $\mathcal{B}$. In this sense the program can be shown to be complete.

In extension we are able to show that the results remain correct if we use certain non-deterministic conditionals (for all clauses must hold essentially that the consistency of two guards implies the equivalence of the respective bodies). Moreover, we can prove for the considered programs that if the input becomes determined, there is no suspending computation left for these programs (a kind of liveness criterion). Due to lack of space this is not subject of this paper.

3 The Calculus

In this section we introduce a calculus providing for an operational semantics of the concurrent constraint language considered in this paper. This language is a subset of the language Oz developed at the DFKI. Ignoring the order of reduction steps leads to dropping the difference between expressions (static) and actors (dynamic) as described in Section 2.2.[3] Because of the reflecting operators, we need to represent the blackboard explicitly on which constraints and abstractions are written.

3.1 The Constraint System $\mathcal{FD}$

$\tilde{x}, \tilde{y}, \ldots$ denote finite, possibly empty sequences of variables. We write $\bot$ and $\top$ for the truth-constant "false" and "true", respectively. Conjunction is assumed to be associative and commutative with $\top$ as identity.

In the previous section we have assumed domains to be a subset of the finite set $\{\mathit{Inf}, \ldots, \mathit{Sup}\}$ (since this is what is usually implemented for finite domain reasoning). Here we consider a more general approach. Let $\mathcal{S}$ denote a set of set-expressions E closed under $\cap$, i.e., $E ::= D \mid E \cap E$, where $\cap$ is assumed to be associative, commutative and idempotent and D are primitive set-expressions. We assume an interpretation $\mathcal{I}$ to exist such that $\mathcal{I} : E \to 2^{\mathbb{Z}}$, $(E_1 \cap E_2)^{\mathcal{I}} = E_1^{\mathcal{I}} \cap E_2^{\mathcal{I}}$, where $\mathbb{Z}$ is the set of integers. We consider only those set expressions E such that it is decidable whether $E^{\mathcal{I}} = \emptyset$, an x is an element of $E^{\mathcal{I}}$, $E^{\mathcal{I}} = \mathbb{Z}$, and whether $E_1^{\mathcal{I}} \subseteq E_2^{\mathcal{I}}$.

[3]Observe that we do not assume the reduction-strategy to be fair. There is no order on the reduction steps beside the one imposed by conditionals. Since we prove the virtual constraint to terminate, fairness is not subject of this paper. If we consider arbitrary expressions in the initial state, we have to consider notions of fairness.

x, y, z, u, v	:		*variable*
p, q	:		*defined predicate*
ϕ, ψ	::=	$\bot \mid \top \mid x \doteq y \mid x \dot{\in} E \mid \phi \wedge \psi$	*constraint*
π	::=	$p{:}\, \tilde{y}/\sigma \mid \pi_1 \wedge \pi_2 \mid \top$	*abstraction* ($\tilde{y}$ *linear*)
β	::=	$\phi \mid \pi \mid \beta_1 \wedge \beta_2$	*blackboard*
α	::=	$p\tilde{y}$	*application*
		if δ **else** σ **fi**	*conditional*
		$\min(xy)$	*minimum*
		$\max(xy)$	*maximum*
		$\alpha_1 \wedge \alpha_2$	*composition*
		$\top$	*truth*
δ	::=	$\omega \mid \delta_1 \vee \delta_2 \mid \bot$	*collection*
ω	::=	$\exists \tilde{x}(\phi$ **then** $\sigma)$	*clause*
σ, τ, μ	::=	$\beta \mid \alpha \mid \exists x \sigma \mid \sigma_1 \wedge \sigma_2$	*expression*
γ	::=	$\beta \diamond \sigma \mid \exists x \gamma$	*state*

Figure 2: Syntax for the calculus

We define the following first-order language with equality. Every $\dot{\in} E$, $E \in \mathcal{S}$, is a unary predicate. The equality symbol $\doteq$ is a binary predicate that is always interpreted as identity. There is no function-symbol, and there is no predicate symbol other than the ones above.

The constraint system $\mathcal{FD}$ consists of the infinite signature $\Sigma = \{\dot{\in} E | E \in \mathcal{S}\}$ of unary predicate symbols and the structure $\mathcal{A}$ over Σ defined as follows. The universe of $\mathcal{A}$ consists of the integers $\mathbb{Z}$. $x \in \dot{\in}^{\mathcal{A}} E$ iff x is an element of $E^{\mathcal{I}}$. For convenience we will write $x \dot{\in} E$ for $\dot{\in} E(x)$ (called membership-constraints).

3.2 Syntax

The abstract syntax of the calculus with respect to the underlying constraint system $\mathcal{FD}$ is shown in Figure 2. It is parameterized over an additional alphabet of distinct predicate symbols, called defined predicate symbols. The translation from abstract syntax to the concrete one used in Section 2.2 should be obvious.

A state consists of two parts. These parts are separated by the symbol $\diamond$. The right part, called the expression-part, contains among others the constraints and abstractions, which are not made 'visible' so far (elaborated). The blackboard is the left part of a state and contains only constraints and abstractions.

We have two forms of quantification: argumental quantification $p{:}\, \tilde{y}/\sigma$ quantifies the variables $\tilde{y}$ with scope σ, and existential quantification $\exists x \sigma$ quantifies x with scope σ and analogously for states. The free variables of an expression and of a state are defined accordingly.

We assume that an expression contains for no defined predicate more than one abstraction and no abstraction is nested into another abstraction.

3.3 Structural Congruence

The congruence $\equiv_{\mathcal{A}}$ on expressions is a congruence satisfying the following axioms.

1. $\sigma \equiv \sigma'$ if σ and σ' are equal up to renaming of variables
2. $\wedge$ is associative, commutative and satisfies $\sigma \wedge \top \equiv \sigma$
3. $\vee$ is associative, commutative and satisfies $\sigma \vee \bot \equiv \sigma$
4. $(\exists x\sigma) \wedge \tau \equiv \exists x(\sigma \wedge \tau)$ if x does not occur free in τ
5. $\exists x \exists y \sigma \equiv \exists y \exists x \sigma$
6. $\exists x \exists y \gamma \equiv \exists y \exists x \gamma$
7. $x \doteq y \wedge \beta \diamond \sigma \equiv x \doteq y \wedge \beta[y/x] \diamond \sigma[y/x]$ if y is free for x in β, and σ
8. $\phi \equiv \psi$ if $\phi \models\!\!\!\dashv_{\mathcal{A}} \psi$.

3.4 Reduction

The reduction relation $\rightarrow$ of the calculus is the least relation on states satisfying the inference rules

$$\frac{\gamma \rightarrow \gamma'}{\exists x\gamma \rightarrow \exists x\gamma'} \qquad \frac{\gamma_1 \equiv_{\mathcal{A}} \gamma_2 \quad \gamma_2 \rightarrow \gamma_3 \quad \gamma_3 \equiv_{\mathcal{A}} \gamma_4}{\gamma_1 \rightarrow \gamma_4}$$

and the following axioms (called reduction rules).

1. $p : \tilde{z}/\sigma \wedge \beta \diamond p\tilde{y} \wedge \tau \rightarrow p : \tilde{z}/\sigma \wedge \beta \diamond \sigma[\tilde{y}/\tilde{z}] \wedge \tau$ if $\tilde{y}$ and $\tilde{z}$ are disjoint, of equal length and $\tilde{y}$ free for $\tilde{z}$ in σ (unfolding)
2. $\beta \diamond \textbf{if } \exists\tilde{x}(\psi \textbf{ then } \tau) \vee \delta \textbf{ else } \mu \textbf{ fi} \wedge \sigma \rightarrow \beta \diamond \textbf{if } \delta \textbf{ else } \mu \textbf{ fi} \wedge \sigma$ if $\beta \equiv_{\mathcal{A}} \phi \wedge \pi$ and $\phi \models_{\mathcal{A}} \neg\exists\tilde{x}\psi$
3. $\beta \diamond \textbf{if } \exists\tilde{x}(\psi \textbf{ then } \tau) \vee \delta \textbf{ else } \mu \textbf{ fi} \wedge \sigma \rightarrow \beta \diamond \exists\tilde{x}(\psi \wedge \tau) \wedge \sigma$ if $\beta \equiv_{\mathcal{A}} \phi \wedge \pi$ and $\phi \models_{\mathcal{A}} \exists\tilde{x}\psi$
4. $\beta \diamond \textbf{if } \bot \textbf{ else } \mu \textbf{ fi} \wedge \sigma \rightarrow \beta \diamond \mu \wedge \sigma$
5. $\beta \diamond \exists x\sigma \rightarrow \exists x(\beta \diamond \sigma)$ if $x \notin \mathcal{V}(\beta)$
6. $\beta \diamond \pi \wedge \sigma \rightarrow \pi \wedge \beta \diamond \sigma$ if $\pi \not\equiv_{\mathcal{A}} \top$
7. $\beta \diamond \phi \wedge \sigma \rightarrow \phi \wedge \beta \diamond \sigma$ if $\phi \not\models\!\!\!\dashv_{\mathcal{A}} \top$ and $\beta \wedge \phi$ consistent; if $\beta \wedge \phi$ is inconsistent, the computation space is failed
8. $\beta \diamond \min(xy) \wedge \sigma \rightarrow \beta \diamond y \doteq m \wedge \sigma$ where $\beta \equiv_{\mathcal{A}} \phi \wedge \pi$ and $m = \max(n \mid \phi \models_{\mathcal{A}} x \geq n)$
9. $\beta \diamond \max(xy) \wedge \sigma \rightarrow \beta \diamond y \doteq m \wedge \sigma$ where $\beta \equiv_{\mathcal{A}} \phi \wedge \pi$ and $m = \min(n \mid \phi \models_{\mathcal{A}} x \leq n)$.

Rules 6 and 7 elaborate abstractions and constraints. Rules 8 and 9 are reflecting the current blackboard. If the blackboard contains more information, the result of applying these rules may change. They make known current information about variables on the blackboard. Observe that it is not allowed to reduce in the body of abstractions and conditionals and in the else-part of conditionals. These positions are called protected positions. They provide us with control to prevent non-terminating computation.

A final state is a state γ such that no more reduction is possible modulo the congruence, i.e., $\forall\gamma'\ \gamma \equiv_{\mathcal{A}} \gamma' : \neg\exists\gamma''\gamma' \rightarrow \gamma''$.

The state for starting computation has an empty blackboard, i.e., $\top \diamond \sigma$. For an example we start with the state

$$\top \diamond \exists x \exists y (x \dot\in \{3,5\} \wedge \texttt{if } x \dot\in \{3,5,6\} \texttt{ then } y \dot\in \{6,9\} \texttt{ else } \mu \texttt{ fi}) :$$

$$\begin{array}{ll}
 & \top \diamond \exists x \exists y (x \dot\in \{3,5\} \wedge \texttt{if } x \dot\in \{3,5,6\} \texttt{ then } y \dot\in \{6,9\} \texttt{ else } \mu \texttt{ fi}) \\
\rightarrow & \exists x \exists y (\top \diamond x \dot\in \{3,5\} \wedge \texttt{if } x \dot\in \{3,5,6\} \texttt{ then } y \dot\in \{6,9\} \texttt{ else } \mu \texttt{ fi}) \\
\rightarrow & \exists x \exists y (x \dot\in \{3,5\} \wedge \top \diamond \texttt{if } x \dot\in \{3,5,6\} \texttt{ then } y \dot\in \{6,9\} \texttt{ else } \mu \texttt{ fi}) \\
\equiv_{\mathcal{A}} & \exists x \exists y (x \dot\in \{3,5\} \diamond \texttt{if } x \dot\in \{3,5,6\} \texttt{ then } y \dot\in \{6,9\} \texttt{ else } \mu \texttt{ fi}) \\
\equiv_{\mathcal{A}} & \exists x \exists y (x \dot\in \{3,5\} \diamond \texttt{if } x \dot\in \{3,5,6\} \texttt{ then } y \dot\in \{6,9\} \texttt{ else } \mu \texttt{ fi} \wedge \top) \\
\rightarrow & \exists x \exists y (x \dot\in \{3,5\} \diamond y \dot\in \{6,9\} \wedge \top) \\
\rightarrow & \exists x \exists y (y \dot\in \{6,9\} \wedge x \dot\in \{3,5\} \diamond \top)
\end{array}$$

4 Verification of the Virtual Constraint

In this section we show the proof techniques and prove correctness, termination and completeness of the considered virtual constraint $x \leq' y$. Due to space restrictions we only sketch the proofs. The details can be read in a forthcoming report. Observe that we first consider only deterministic conditionals; in Section 4.4 we also consider non-deterministic conditionals.

4.1 Correctness

In this section we prove the correctness of Program 2.1 with respect to an intended model. Every state without non-deterministic conditionals is translated into a first-order formula by the mapping $[\![\,]\!]$ defined as follows. Composition and $\diamond$ translate to conjunction, quantification to existential quantification, application, abstraction and deterministic conditional as follows:

$$[\![p : \tilde{x}/\sigma]\!] := \forall \tilde{x}(p(\tilde{x}) \leftrightarrow [\![\sigma]\!])$$

$$[\![p\tilde{x}]\!] := p(\tilde{x})$$

$$[\![\texttt{if } \exists \tilde{x}(\phi \texttt{ then } \sigma) \texttt{ else } \tau \texttt{ fi}]\!] := \exists \tilde{x}(\phi \wedge [\![\sigma]\!]) \;\vee\; (\neg \exists \tilde{x} \phi) \wedge [\![\tau]\!].$$

We say that a transformation of computation states by congruence or reduction rules is an equivalence transformation if the corresponding first order formulas are equivalent with respect to a given model.

Next we define a persistent extension of the structure $\mathcal{A}$, also called $\mathcal{A}$, with signature $\Sigma = \{\dot\in E, \leq, \geq, <, >, 0, \perp 1, 1, \perp 2, 2, \ldots\}$ and the universe being the integers. $\leq^{\mathcal{A}}$ etc. are the relations on the integers as usual.

We now define a persistent extension $\mathcal{B}$ of $\mathcal{A}$: the intended model. The declarative semantics of the predicates and actors are shown in Table 1. Observe that we have $x \dot\in \{v, v+1, \ldots\} \models\!\mid_{\mathcal{A}} x \geq v \models\!\mid_{\mathcal{A}} v \leq x$ and analogously for $x \dot\in \{\ldots, v \perp 1, v\}$, i.e., these are unary constraints of $\mathcal{FD}$.

The following proposition states that congruence transformations are equivalence preserving in all persistent extensions of $\mathcal{A}$.

Proposition 4.1 *Let $\mathcal{C}$ be a persistent extension of $\mathcal{A}$. Then, $\gamma \equiv_{\mathcal{A}} \gamma' \Rightarrow [\![\gamma]\!] \models\!\mid_{\mathcal{C}} [\![\gamma']\!]$ holds.*

extension	semantics
$(xv) \in \min^{\mathcal{B}}$	$v \leq^{\mathcal{A}} x$
$(xv) \in \max^{\mathcal{B}}$	$x \leq^{\mathcal{A}} v$
$(xyv) \in \mathit{propLow}^{\mathcal{B}}$	$x \leq^{\mathcal{A}} y \wedge v \leq^{\mathcal{A}} x$
$(xyv) \in \mathit{propUp}^{\mathcal{B}}$	$x \leq^{\mathcal{A}} y \wedge y \leq^{\mathcal{A}} v$
$(xv) \in \mathit{greater}^{\mathcal{B}}$	$x >^{\mathcal{A}} v$
$(xv) \in \mathit{gec}^{\mathcal{B}}$	$x \geq^{\mathcal{A}} v$
$(xv) \in \mathit{less}^{\mathcal{B}}$	$x <^{\mathcal{A}} v$
$(xv) \in \mathit{lec}^{\mathcal{B}}$	$x \leq^{\mathcal{A}} v$
$(xy) \in \leq'^{\mathcal{B}}$	$x \leq^{\mathcal{A}} y$

Table 1: Persistent Extension $\mathcal{B}$

Proof: Obvious.□

The following lemma states that the chosen semantics is indeed a correct one: the first-order formulas resulting from the bodies of the abstractions are logically equivalent to the chosen semantics. Therefore, unfolding is an equivalence transformation.

Lemma 4.2 *If $\gamma \to \gamma'$ by rule 1 (unfolding), then $[\![\gamma]\!] \models\!\mid_{\mathcal{B}} [\![\gamma']\!]$ holds, if computation starts in a state containing the procedure definitions of Program 2.1, a finite set of membership-constraints and an application of the procedure $\leq'$.*

Proof: We prove the claim for *propLow*; an analogous argumentation holds for *propUp*. For an application of rule 1 we consider the reduction $\gamma \to \gamma'$ where $\gamma = \mathit{propLow} : \tilde{x}/\sigma \wedge \beta \diamond \mathit{propLow}\, xy\underline{x} \wedge \tau$ and $\gamma' = \mathit{propLow} : \tilde{x}/\sigma \wedge \beta \diamond \sigma[xy\underline{x}/\tilde{x}] \wedge \tau$. Translating γ we obtain

$$[\![\gamma]\!] = [\![\mathit{propLow} : \tilde{x}/\sigma]\!] \wedge [\![\beta]\!] \wedge x \leq y \wedge \underline{x} \leq x \wedge [\![\tau]\!]$$

and by translating γ' we obtain

$$[\![\gamma']\!] = [\![\mathit{propLow} : \tilde{x}/\sigma]\!] \wedge [\![\beta]\!] \wedge [\![\texttt{if } x > \underline{x} \texttt{ then } \ldots \texttt{ else } \top\texttt{fi}]\!] \wedge [\![\tau]\!].$$

For the proof we use the translation of the conditional into the disjunction

$$x > \underline{x} \;\wedge\; \exists u(u \leq x \;\wedge\; u \leq y \;\wedge\; x \leq y) \;\vee\; x \leq \underline{x}. \tag{1}$$

Assume the if-case, i.e., $x > \underline{x}$. Then,

$$(1) \models\!\mid_{\mathcal{B}} x > \underline{x} \;\wedge\; \exists u(u \leq x \;\wedge\; u \leq y \;\wedge\; x \leq y) \models\!\mid_{\mathcal{B}} x > \underline{x} \;\wedge\; x \leq y.$$

This proves the equivalence of $[\![\gamma]\!]$ and $[\![\gamma']\!]$ because of the assumption. Assume the else-case, i.e., $x \leq \underline{x}$. The disjunction is equivalent to

$$x \leq \underline{x}.$$

Now we have to consider also β and τ of γ'. Because of the use of local quantification in the virtual constraint, $\underline{x}$ can only be computed in the body of $\leq'$ or in the body of the conditional of *propLow*. In the first case, the constraint $\underline{x} \leq x \wedge y \geq \underline{x}$ was imposed (up to renaming). Thus,

$[\![\gamma']\!] \models_{\mathcal{B}} x \doteq \underline{x} \wedge \underline{x} \leq y$. An analogous argumentation holds in the second case. This is proves the equivalence of $[\![\gamma]\!]$ and $[\![\gamma']\!]$ because of the assumption and, thus, the claim for *propLow*.

For procedure $\leq'$ we prove

$$\begin{array}{rll} x \leq y & \models\!\mid_{\mathcal{B}} & x \leq y \ \wedge \ \exists \underline{x}\overline{y}(\underline{x} \leq x \ \wedge \ \underline{x} \leq y \ \wedge \ x \leq \overline{y} \ \wedge \ y \leq \overline{y}) \\ & \models\!\mid_{\mathcal{B}} & \exists \underline{x}\overline{y}(x \leq y \ \wedge \ \underline{x} \leq x \ \wedge \ \underline{x} \leq y \ \wedge \ x \leq \overline{y} \ \wedge \ y \leq \overline{y}) \\ & \models\!\mid_{\mathcal{B}} & [\![\exists \underline{\mathrm{x}}\ \overline{\mathrm{y}}\ \ldots \mathit{propUp}\, \mathrm{xy}\overline{\mathrm{y}}]\!]. \end{array}$$

This proves the lemma. □

Next we have to prove that reduction rules 8 and 9 are equivalence transformations in the context of Program 2.1. That e.g. rule 8 is not equivalence preserving in general becomes clear by considering the following example: $x \geq 5\ y \geq 3 \diamond \min(xu)$ reduces to $x \geq 5\ y \geq 3 \diamond u = 5$ but their translations $x \geq 5 \ \wedge \ y \geq 3 \ \wedge \ u \leq x$ and $x \geq 5 \ \wedge \ y \geq 3 \ \wedge \ u \doteq 5$ are not equivalent.

Lemma 4.3 *If $\gamma \to \gamma'$ by reduction rules 8 or 9, then $[\![\gamma]\!] \models\!\mid_{\mathcal{B}} [\![\gamma']\!]$ holds, if computation starts in a state containing the procedure definitions of Program 2.1, a finite set of membership-constraints and an application of the procedure $\leq'$.*

Proof: The state before and after application of rule 8 (9) is translated into a first order formula. We consider the case where rule 8 is applied in the reduction $\gamma \to \gamma'$. Assume that the constraint on the blackboard is contained in ϕ with $\phi \models_{\mathcal{B}} x \geq n, n = \max(m \mid \phi \models_{\mathcal{B}} x \geq m)$. Observe that at most one application of *propLow* occurs in a state. Furthermore, we need not to consider the case, where *propLow* is unfolded to its body, because we have proved in Lemma 4.2 that unfolding is an equivalence transformation (and we now consider formulas only). The equivalence between the two formulas $[\![\gamma]_F$ and $[\gamma']\!]$ can be shown as follows (τ contains further expressions of the translated state).

$$\begin{array}{ll} & \exists \tilde{x}(\phi \wedge \tau \wedge \exists u([\![\min(xu)]\!] \wedge y \geq u \wedge [\![\mathit{propLow}\, \mathrm{xy}u]\!])) \\ \models\!\mid_{\mathcal{B}} & \exists \tilde{x}(\phi \ \wedge \ \tau \ \wedge \ \exists u(u \leq x \ \wedge \ y \geq u \ \wedge \ x \leq y \ \wedge \ u \leq x)) \\ \models\!\mid_{\mathcal{B}} & \exists \tilde{x}(\phi \ \wedge \ \tau \ \wedge \ \exists u(u \leq x \ \wedge \ y \geq u) \ \wedge \ x \leq y) \\ \models\!\mid_{\mathcal{B}} & \exists \tilde{x}(\phi \ \wedge \ \tau \ \wedge \ x \leq y) \\ \models\!\mid_{\mathcal{B}} & \exists \tilde{x}(\phi \ \wedge \ \tau \ \wedge \ x \leq y \ \wedge \ n \leq x) \qquad \mathit{since}\ \phi \models_{\mathcal{B}} \mathrm{x} \geq n \\ \models\!\mid_{\mathcal{B}} & \exists \tilde{x}(\phi \ \wedge \ \tau \ \wedge \ x \leq y \ \wedge \ n \leq x \ \wedge \ n \leq y) \\ \models\!\mid_{\mathcal{B}} & \exists \tilde{x}(\phi \ \wedge \ \tau \ \wedge \ \exists u(u \doteq n \ \wedge \ y \geq u \ \wedge \ x \leq y \ \wedge \ u \leq x)) \\ \models\!\mid_{\mathcal{B}} & \exists \tilde{x}(\phi \ \wedge \ \tau \ \wedge \ \exists u(u \doteq n \ \wedge \ y \geq u \ \wedge \ [\![\mathit{propLow}\, \mathrm{xy}u]\!])) \end{array}$$

An analogous argumentation[4] holds for rule 9 in *propUp* and 8 / 9 in procedure $\leq'$. □

Since it is obvious that the other reduction rules are equivalence preserving, Theorem 4.4 follows.

Theorem 4.4 $\gamma \to \gamma' \ \Rightarrow \ [\![\gamma]\!] \models\!\mid_{\mathcal{B}} [\![\gamma']\!]$ *for all reduction steps of Section 3.4, if computation starts in a state containing the procedure definitions of Program 2.1, a finite set of membership-constraints and an application of the procedure $\leq'$.*

4.2 Termination

Termination means that no more reduction rule is applicable. Observe that in case of an infinite number of reduction steps, there must be an infinite number of unfolding steps. Since only the procedures *propLow* and *propUp* contain recursive calls we only have to analyze these.

[4]The careful reader may wonder why the semantics of $\min(xu)$ is chosen as $u \leq x$. In the considered example there are also other semantics possible. But this does not hold for virtual constraints like $x + y = z$ where the semantics of $\min$ is exploited to prove the equivalence of formulas (for more details see the forthcoming report).

Theorem 4.5 *Program 2.1 terminates if computation starts in a state containing the procedure definitions of Program 2.1, a finite set of membership-constraints and an application of the procedure $\leq'$.*

Proof: The proof uses a lexicographic ordering based on the number of not yet unfolded applications of *propLow/propUp* contained in the computation state, the number of variables without lower or upper bounds, and an integer reflecting the current minimum and maximum of variables' domains. The first number is decremented if a conditional reduces to an else-case. The other numbers are lowered if a conditional reduces to an if-case. □

4.3 Completeness

Let ϕ be a finite set of membership-constraints, π the abstractions of a virtual constraint p with the declarative specification ψ, i.e., $[\![p\tilde{x}]\!] = \psi$, and $p\tilde{x}$ an application of the virtual constraint. Let $\top \diamond p\tilde{x} \wedge \pi \wedge \phi$ be the starting state. A virtual constraint π is called complete if in case the computation space is not failed by reductions inititated in the starting state, then $\psi \wedge \phi$ is satisfiable. If the computation space is not failed, we can explicitly construct a valuation satisfying $\psi \wedge \phi$. Since we have shown the reductions to be equivalence transformations, the domains on the final blackboard are the largest possible ones serving as a basis for solutions.

Theorem 4.6 *Program 2.1 can be used to decide satisfiability of a constraint $x \leq y$ and a finite set of membership-constraints.*

Proof: One can prove that for a final state $\gamma = \exists\tilde{x}(\phi \wedge \pi \diamond \sigma)$

$$\forall \underline{x}(\phi \rightarrow \underline{x} \leq x) \rightarrow (\phi \rightarrow \underline{x} \leq y) \text{ and } \forall \overline{y}(\phi \rightarrow y \leq \overline{y}) \rightarrow (\phi \rightarrow x \leq \overline{y})$$

are valid. Assume for the final constraint-part

$$n = \max(k|\phi \models_{\mathcal{B}} x \geq k) \text{ and } m = \max(k|\phi \models_{\mathcal{B}} y \geq k).$$

A valuation for $x \leq y$ can be obtained by chosing $x_v = n, y_v = m$. Now we consider the case that x has no lower bound in ϕ. We choose for x_v the value $min(\min(n|n \leq z \in \phi), \min(n \perp 1|z \neq n \in \phi))$. If y has a lower bound y_v, $x_v \leq^{\mathcal{B}} y_v$ holds. If y has also no lower bound, take x_v for y_v.[5] □

4.4 Non-deterministic Conditionals

Finally, we consider non-deterministic conditionals. Since by first-order formulas one cannot express non-determinism, it is not possible in general to give a translation for conditionals into first-order formulas. But if two guards are consistent, i.e., their conjunction is satisfiable, and the meaning of their body in conjunction with the respective guard are equivalent, it does not matter which of the possible clauses of the conditional is chosen. This can be formalized by the following theorem.

Theorem 4.7 *Let $\mathcal{C}$ be an arbitrary structure. If for a conditional*

[5] Observe that this argumentation can be generalized to the case of more than one virtual constraint $\leq'$.

```
if $\tilde{x}_1$ in $\psi_1$ then $\tau_1$
[] $\tilde{x}_2$ in $\psi_2$ then $\tau_2$
...
[]  $\tilde{x}_n$ in $\psi_n$ then $\tau_n$
else $\tau_{n+1}$
fi
```

holds that

$$\mathcal{C} \models \exists\tilde{x}_i\psi_i \wedge \exists\tilde{x}_j\psi_j \;\rightarrow\; (\exists\tilde{x}_i(\psi_i \wedge [\![\tau_i]\!]) \leftrightarrow \exists\tilde{x}_j(\psi_j \wedge [\![\tau_j]\!]))$$

for all i, j*, then the conditional has the declarative semantics*

$$\exists\tilde{x}_1(\psi_1 \wedge [\![\tau_1]\!]) \;\vee\; \ldots \;\vee\; \exists\tilde{x}_n(\psi_n \wedge [\![\tau_n]\!]) \;\vee\; (\neg\exists\tilde{x}_1\psi_1) \wedge \ldots \wedge (\neg\exists\tilde{x}_n\psi_n) \wedge [\![\tau_{n+1}]\!]$$

and the reduction rules 2 *and* 3 *are equivalence transformations.*

It can be shown that the virtual constraint $\leq'$ can be implemented using one non-deterministic conditional. This program has the property that a final state contains no suspending conditional if the inital state entails that one of input variables of the constraint is determined.

5 Outlook

In this paper we have proved the correctness, termination and completeness of an algorithm in the area of finite domain reasoning. The proofs rely on the translation of computation states of a concurrent constraint language into first-order formulas. To this aim we define a model by giving the procedures occurring in the definition of the virtual constraint $\leq'$ a declarative semantics (and also the reflecting operators). We then have proved that for a reduction of computation states $\gamma \rightarrow \gamma'$ the obtained first order formulas of γ and γ' are equivalent with respect to this model.

The described approach succeeds in verifying the program under consideration. Nevertheless, the treatment of additional expressions in the initial state and parts of the proofs show that there are two tracks of verification. One track exploits the first-order semantics for correctness and completeness (shown in this paper). The other track exploits the operational semantics for statements in the context of computation. We aim to bring together these two tracks. We have to design a logic of operational behavior of expressions in Oz. Properties like correctness, termination and completeness must be expressible in this logic. The logic must account computational contexts of Oz expressions, which can be seen as the counterpart of invariants in the setting of concurrent reduction systems. Modal logic seems to be a natural candidate for such a logic (see for example [MP92]). For the future, we want to design a calculus for this logic allowing to prove in a formal system operational properties of Oz expressions.

Acknowledgement: I would like to thank Ralf Treinen and Gert Smolka for valuable discussions. The research reported in this paper has been supported by the Bundesminister für Forschung und Technologie (FTZ-ITW-9105).

References

[dBGMP94] F.S. de Boer, M. Gabbrielli, E. Marchori, and C. Palamidessi. Proving concurrent constraint programs correct. In **Proceedings of the ACM Symposium on Principles of Programming Languages**, 1994.

[DC93] D. Diaz and P. Codognet. A minimal extension of the WAM for clp(FD). In **Proceedings of the International Conference on Logic Programming**, pages 774–790, Budapest, Hungary, 1993. MIT Press.

[DVS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In **Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88**, pages 693–702, Tokyo, Japan, December 1988.

[HMM+94] M. Henz, M. Mehl, M. Müller, T. Müller, J. Niehren, R. Scheidhauer, C. Schulte, G. Smolka, R. Treinen, and J. Würtz. The Oz Handbook. Research Report RR-94-09, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994. Available through anonymous ftp from `duck.dfki.uni-sb.de`.

[HSW93] M. Henz, G. Smolka, and J. Würtz. Oz - A programming language for multi-agent systems. In **Proceedings of the 13th International Joint Conference on Artificial Intelligence**, pages 404–409, Chambéry, France, August 1993. Morgan Kaufmann.

[Mac77] A. K. Mackworth. Consistency in networks of relations. **Artificial Intelligence**, 8:99–118, 1977.

[Mil91] R. Milner. The polyadic π-calculus: A tutorial. ECS-LFCS Report Series 91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, October 1991.

[MP92] Z. Manna and A. Pnueli. **The temporal logic of reactive and concurrent systems**. Springer Verlag, 1992.

[SHW93] G. Smolka, M. Henz, and J. Würtz. Object-oriented concurrent constraint programming in Oz. Research Report RR-93-16, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, April 1993. Will appear in: P. van Hentenryck and V. Saraswat (eds.), Principles and Practice of Constraint Programming, The MIT Press, Cambridge, Mass.

[Smo94] G. Smolka. A calculus for higher-order concurrent constraint programming with deep guards. Research Report RR-94-03, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, February 1994.

[SR90] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In **Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages**, pages 232–245, San Francisco, CA, January 1990.

[SSW94] C. Schulte, G. Smolka, and J. Würtz. Encapsulated search and constraint programming in Oz. In **Second Workshop on Principles and Practice of Constraint Programming**, pages 116–129, Orcas Island, Washington, USA, 2-4 May 1994.

[VDT92] P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. **Artificial Intelligence**, 57:291–321, 1992.

[VSD93] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). Report CS-93-02, Brown University, January 1993.