Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research
Report
RR-93-16

Object-Oriented Concurrent Constraint Programming in Oz

Gert Smolka, Martin Henz, Jörg Würtz

April 1993

Deutsches Forschungszentrum für Künstliche Intelligenz
GmbH

Postfach 20 80
D-6750 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, SEMA Group, and Siemens. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- ❑ Intelligent Engineering Systems
- ❑ Intelligent User Interfaces
- ❑ Computer Linguistics
- ❑ Programming Systems
- ❑ Deduction and Multiagent Systems
- ❑ Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Friedrich J. Wendl
Director

Object-Oriented Concurrent Constraint Programming in Oz

Gert Smolka, Martin Henz, Jörg Würtz

DFKI-RR-93-16

This work has been supported by a grant from the Bundesminister für Forschung und Technologie (FKZ ITW-9105) and by the ESPRIT basic research project 7195 (ACCLAIM).

Object-Oriented Concurrent Constraint Programming in Oz

Gert Smolka, Martin Henz, Jörg Würtz

German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D-6600 Saarbrücken, Germany
E-mail: {smolka, henz, wuertz}@dfki.uni-sb.de

Abstract

Oz is an experimental higher-order concurrent constraint programming system under development at DFKI. It combines ideas from logic and concurrent programming in a simple yet expressive language. From logic programming Oz inherits logic variables and logic data structures, which provide for a programming style where partial information about the values of variables is imposed concurrently and incrementally. A novel feature of Oz is that it accommodates higher-order programming without sacrificing that denotation and equality of variables are captured by first-order logic. Another new feature of Oz is constraint communication, a new form of asynchronous communication exploiting logic variables. Constraint communication avoids the problems of stream communication, the conventional communication mechanism employed in concurrent logic programming. Constraint communication can be seen as providing a minimal form of state fully compatible with logic data structures.

Based on constraint communication and higher-order programming, Oz readily supports a variety of object-oriented programming styles including multiple inheritance.

Contents

1 Introduction

Oz is an attempt to create a high-level concurrent programming language bringing together the merits of logic and object-oriented programming.

Our starting point was concurrent constraint programming [14], which brings together ideas from constraint and concurrent logic programming. Constraint logic programming [8, 4], on the one hand, originated with Prolog II [5] and was prompted by the need to integrate numbers and data structures in an operationally efficient, yet logically sound manner. Concurrent logic programming [15], on the other hand, originated with the Relational Language [3] and was promoted by the Japanese Fifth Generation Project, where logic programming was conceived as the basic system programming language and thus had to account for concurrency, synchronization and indeterminism. For this purpose, the conventional SLD-resolution scheme had to be replaced with a new computation model based on the notion of committed choice. At first, the new model developed as an ad hoc construction, but finally Maher [11] realized that commitment of agents can be captured logically as constraint entailment. A major landmark in the new field of concurrent constraint programming is AKL [9], the first implemented concurrent constraint language accommodating search and deep guards.

The concurrent constraint model [14] can accommodate object-oriented programming along the lines of Shapiro and Takeuchi's stream-based model for Concurrent Prolog [16, 10]. Unfortunately, this model is intolerably low-level, which becomes fully apparent when one considers inheritance [7]. Vulcan, Polka, and A'UM are attempts to create high-level object-oriented languages on top of concurrent logic languages (see [10] for references). Due to the wide gap these languages have to bridge, they however loose the simplicity and flexibility of the underlying base languages.

Oz avoids these difficulties by extending the concurrent constraint model with the features needed for a high-level object model: a higher-order programming facility and a communication primitive avoiding the clumsiness of stream communication. With these extensions the need for a separate object-oriented language disappears since the base language itself can express objects and inheritance in a concise and elegant way.

The way Oz provides for higher-order programming is unique in that denotation and equality of variables are nevertheless captured by first-order logic only. In fact, denotation of variables and the facility for higher-order programming are completely orthogonal concepts in Oz. This is in contrast to existing approaches to higher-order logic programming [13, 2].

Constraint communication is asynchronous and indeterministic. A communication event replaces two complementary communication tokens with an equality constraint linking the partners of the communication. Constraint communication introduces a minimal form of state that is fully compatible with logic data structures. Efficient implementation of fair constraint communication is straightforward.

The new concepts in Oz cannot be accounted for within the established semantical frameworks. Thus the semantics of Oz is specified by a new mathematical model, called the Oz Calculus, whose technical setup was inspired by the π-calculus [12], a recent foundationally motivated model of concurrency.

The paper is organized as follows. The next section outlines a simplified version of the Oz Calculus. Section 3 shows how the constraint system of Oz accommodates records, which are the congenial data structure for object-oriented programming. Section 4 introduces the concrete language. Section 5 presents one possible style of object-oriented programming in Oz featuring multiple inheritance.

2 The Oz Calculus

The operational semantics of Oz is defined by a mathematical model called the Oz Calculus [17]. In this section we outline a simplified version sufficing for the purposes of this paper.

The basic notion of Oz is that of a computation space. A computation space consists of a number of agents connected to a blackboard (see Fig. 1). Each agent reads the blackboard and reduces once the blackboard contains the information it is waiting for. The information on the blackboard increases monotonically. When an agent reduces, it may put new information on the blackboard and create

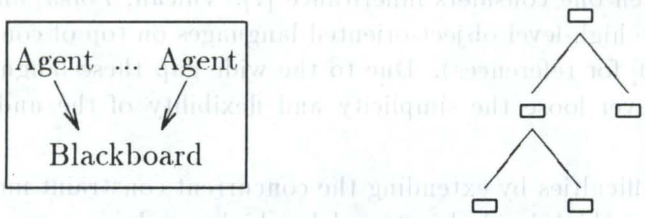

Figure 1: The blackboard metaphor

new agents. Agents themselves may have one or several local computation spaces. Hence the entire computation system is a tree-like structure of computation spaces (see Fig. 1).

The agents of a computation space disappear as soon as they reduce. We will see later how one can express long-lived agents with persistent identity.

Formally, a computation state is an expression σ according to Fig. 2. (If ξ is a syntactic category, $\overline{\xi}$ denotes a possibly empty sequence $\xi \ldots \xi$.) Constraints, abstractions and communication tokens reside on the blackboard. Applications and conditionals are agents. Composition and quantification are the glue assembling agents and blackboard items into a computation space. Quantification introduces local variables. Abstractions may be seen as procedure definitions and applications as procedure calls.

x, y, z	:	*variables*	
σ, τ, μ	$::=$	ϕ	*constraint*
		$x{:}\overline{y}/\sigma$	*abstraction*
		$x\,!\,y$	*put token*
		$x\,?\,y$	*get token*
		$x\overline{y}$	*application*
		if $\omega\ \ldots\ \omega$ **else** σ	*conditional*
		$\sigma \wedge \tau$	*composition*
		$\exists x \sigma$	*quantification*
ω	$::=$	$\exists\overline{x}\,(\sigma$ **then** $\tau)$	*clause*
ϕ, ψ	$::=$	$\bot \mid \top \mid s \doteq t \mid r(\overline{s}) \mid \phi \wedge \psi$	

Figure 2: Expressions of the Oz Calculus

The clauses of a conditional are unordered. Their guards, i.e., σ in $\exists\overline{x}\,(\sigma$ **then** $\tau)$, constitute local computation spaces. Note that any expression can be taken as a guard; one speaks of a *flat* guard if the guard is a constraint.

There are two variable binders: quantification $\exists x\sigma$ binds x with scope σ, and abstraction $x{:}\overline{y}/\sigma$ binds the variables in $\overline{y}$ with scope σ. Free variables of an expression are defined accordingly.

Computation is defined as reduction (i.e., rewriting) of expressions. A reduction step is performed by applying a reduction rule to a subexpression satisfying the application conditions of the rule. There is no backtracking. Control is provided by the provision that reduction rules must not be applied to *mute* subexpressions, i.e., subexpressions that occur within bodies of clauses, else parts of conditionals, or bodies of abstractions. It is up to the implementation which non-mute subexpression is rewritten by which applicable rule.

Reduction "$\sigma \to \tau$" is defined modulo structural congruence "$\sigma \equiv \tau$" of expressions, that is, satisfies the inference rule

$$\frac{\sigma \equiv \sigma' \quad \sigma' \to \tau' \quad \tau' \equiv \tau}{\sigma \to \tau}.$$

Structural congruence is an abstract equality for computation states turning them from purely syntactic objects into semantic objects. Structural congruence provides for associativity and commutativity of composition, renaming of bound variables, quantifier mobility

$$\exists x\sigma \wedge \tau \equiv \exists x(\sigma \wedge \tau) \quad \text{if } x \text{ does not occur free in } \tau,$$

constraint simplification, and information propagation from global blackboards to local blackboards.

2.1 Constraints

Constraints (ϕ,ψ in Figure 2) are formulas of first-order predicate logic providing for data structures. Logical conjunction of constraints coincides with composition of expressions. Constraints express partial information about the values of variables. The semantics of constraints is defined logically by a first-order theory Δ and is imposed on the calculus by the congruence law

$$\phi \equiv \psi \qquad \text{if } \Delta \models \phi \leftrightarrow \psi.$$

This law closes the blackboard under entailed constraints (since $\Delta \models \phi \rightarrow \psi$ iff $\Delta \models \phi \leftrightarrow \phi \wedge \psi$). The congruence law

$$x \doteq y \wedge \sigma \;\equiv\; x \doteq y \wedge \sigma[y/x] \qquad \text{if } y \text{ is free for } x \text{ in } \sigma$$

extends equalities on the blackboard to the rest of the computation space ($\sigma[y/x]$ is obtained from σ by replacing every free occurrence of x with y). Equality of variables is strictly first-order: Two variables x, y are equal if the constraints on the blackboard entail $x \doteq y$, and different if the constraints on the blackboard entail $\neg(x \doteq y)$. The information on the blackboard may be insufficient to determine whether two variables are equal or different. Moreover, an inconsistent blackboard entails both $x \doteq y$ and $\neg(x \doteq y)$.

The Anullation Law

$$\exists\overline{x}(\phi \;\wedge\; \overline{y}\!:\overline{\alpha}) \;\equiv\; \top$$

$$\text{if } \Delta \models \exists\overline{x}\,\phi \text{ and } \overline{y} \subseteq \mathcal{L}(\overline{x},\phi), \text{ where}$$
$$\mathcal{L}(\overline{x},\phi) \;:=\; \{y \in \overline{x} \mid \forall z\!:\, \phi \models_\Delta y \doteq z \;\Rightarrow\; z \in \overline{x}\}$$

provides for the deletion of quantified constraints and abstractions not affecting visible variables. $\mathcal{L}(\overline{x},\phi)$ is the set of all variables in $\overline{x}$ that are not equated to variables outside of $\overline{x}$ by the constraint ϕ.

2.2 Application

An application agent $x\overline{y}$ waits until an abstraction for its *link* x appears on the blackboard and then reduces as follows:

$$x\overline{y} \;\wedge\; x\!:\overline{z}/\sigma \;\rightarrow\; \exists\overline{z}\,(\overline{z} \doteq \overline{y} \wedge \sigma) \;\wedge\; x\!:\overline{z}/\sigma$$
$$\text{if } \overline{y} \text{ and } \overline{z} \text{ are disjoint and of equal length.}$$

Note that the blackboard $y\!:\overline{z}/\sigma \wedge x \doteq y$ contains an abstraction for x due to the congruence laws stated above. Since the link x of an abstraction $x\!:\overline{y}/\sigma$ is a variable like any other, abstractions can easily express higher-order procedures. Note that an abstraction $x\!:\overline{y}/\sigma$ does not impose any constraints (e.g., equalities) on its link x.

2.3 Constraint Communication

The semantics of the two communication tokens is defined by the Communication Rule:

$$x\,!\,y \wedge z?\,y \;\rightarrow\; x \doteq z.$$

Application of this rule amounts to an indeterministic transition of the blackboard replacing two complementary communication tokens sharing the same link y with an equality constraint. The Communication Rule is the only rule deleting items from the blackboard. Since agents read only constraints and abstractions, the information visible to agents nevertheless increases monotonically.

2.4 Conditional

It remains to explain the semantics of a conditional agent

$$\textbf{if}\ \exists\overline{x}_1\,(\sigma_1\ \textbf{then}\ \tau_1)\ \cdots\ \exists\overline{x}_n\,(\sigma_n\ \textbf{then}\ \tau_n)\ \textbf{else}\ \mu.$$

The guards σ_i of the clauses are local computation spaces reducing concurrently. For the local computations to be meaningful it is essential that information from global blackboards is visible on local blackboards. This is achieved with the Propagation Law (recall that the clauses are unordered):

$$\pi\ \wedge\ \textbf{if}\ \exists\overline{x}\,(\sigma\ \textbf{then}\ \tau)\ \overline{\omega}\ \textbf{else}\ \mu$$
$$\equiv$$
$$\pi\ \wedge\ \textbf{if}\ \exists\overline{x}\,(\pi \wedge \sigma\ \textbf{then}\ \tau)\ \overline{\omega}\ \textbf{else}\ \mu$$

if π is a constraint or abstraction and
no variable in $\overline{x}$ appears free in π.

Read from top to bottom, the law provides for copying information from global blackboards to local blackboards. Read from bottom to top, the law provides for deletion of local information that is present globally. An example illustrating the application of the Propagation Law in both directions (as well as constraint simplification) is

$$x \doteq 1 \wedge\ \textbf{if}\ (x \doteq 1\ \textbf{then}\ \sigma)\ (x \doteq 2\ \textbf{then}\ \tau)\ \textbf{else}\ \mu$$
$$\equiv\ x \doteq 1\ \wedge\ \textbf{if}\ (\top\ \textbf{then}\ \sigma)\ (\bot\ \textbf{then}\ \tau)\ \textbf{else}\ \mu.$$

The example assumes that the constraint theory entails that 1 and 2 are different.

Operationally, the constraint simplification and propagation laws can be realized with a so-called relative simplification procedure [1]. Relative simplification for the constraint system underlying Oz is investigated in [18].

There are two distinguished forms a guard of a clause may eventually reduce to, called *satisfied* and *failed*. If a guard of a clause is satisfied, the conditional can reduce by committing to this clause:

$$\textbf{if}\ \exists\overline{x}\,(\sigma\ \textbf{then}\ \tau)\ \overline{\omega}\ \textbf{else}\ \mu\ \rightarrow\ \exists\overline{x}\,(\sigma \wedge \tau) \qquad \text{if}\ \exists\overline{x}\sigma \equiv \top.$$

Reduction puts the guard on the global blackboard and releases the body of the clause.

A guard is failed if the constraints on its blackboard are unsatisfiable. If the guard of a clause is failed, the clause is simply discarded:

$$\textbf{if } \exists\overline{x}\,(\bot \wedge \sigma \textbf{ then } \tau)\ \overline{\omega}\textbf{ else } \mu \;\rightarrow\; \textbf{if } \overline{\omega}\textbf{ else } \mu.$$

Thus a conditional may end up with no clauses at all, in which case it reduces to its else part:

$$\textbf{if else } \mu \;\rightarrow\; \mu.$$

The reduction

$$x \doteq 1 \;\wedge\; \textbf{if } (x \doteq 1 \textbf{ then } \sigma)\,(x \doteq 2 \textbf{ then } \tau) \textbf{ else } \mu \quad\rightarrow\quad x \doteq 1 \;\wedge\; \sigma$$

is an example for the application of the first rule, and

$$x \doteq 3 \;\wedge\; \textbf{if } (x \doteq 1 \textbf{ then } \sigma)\,(x \doteq 2 \textbf{ then } \tau) \textbf{ else } \mu \quad\rightarrow^*\quad x \doteq 3 \;\wedge\; \mu$$

is an example employing the other two reduction rules.

2.5 Logical Semantics

The subcalculus obtained by weakening the Anullation Law to

$$\exists\overline{x}\,\phi \;\equiv\; \top \qquad \text{if } \Delta \models \exists\overline{x}\,\phi$$

and disallowing communication tokens and conditionals with more than one clause enjoys a logical semantics, which is obtained by translating expressions into formulas of first-order predicate logic as follows: composition translates to conjunction, quantification to existential quantification, and abstraction, application and conditional translate as follows:

$$\begin{aligned} x{:}\,\overline{y}/\sigma &\;\Longrightarrow\; \forall\overline{y}\,(\mathsf{apply}(x\overline{y}) \leftrightarrow \sigma) \\ x\overline{y} &\;\Longrightarrow\; \mathsf{apply}(x\overline{y}) \\ \textbf{if } \exists\overline{x}\,(\sigma \textbf{ then } \tau) \textbf{ else } \mu &\;\Longrightarrow\; \exists\overline{x}\,(\sigma \wedge \tau) \,\vee\, (\neg\exists\overline{x}\sigma \wedge \mu). \end{aligned}$$

Under this translation, reduction is an equivalence transformation, that is, if $\sigma \rightarrow \tau$ or $\sigma \equiv \tau$, then $\Delta \models \sigma \leftrightarrow \tau$. Moreover, negation can be expressed since $\neg\sigma$ is equivalent to **if** σ **then** $\bot$ **else** $\top$.

2.6 Unique Names

A problem closely related to equality and of great importance for concurrent programming is the dynamic creation of new and unique names. Roughly, one would like to have a construct $\mathsf{gensym}(x)$ such that $\mathsf{gensym}(x) \wedge \mathsf{gensym}(y)$ is congruent to a constraint entailing $\neg(x \doteq y)$. For this purpose we assume that there are infinitely many distinguished constant symbols called *names* such that the constraint theory Δ satisfies:

1. $\Delta \models \neg(a \doteq b)$ for every two distinct names a, b
2. validity of sentences with respect to Δ is invariant under permutation of names.

Now $\mathsf{gensym}(x)$ is modeled as a generalized quantification $\exists a(x \doteq a)$, where the quantified name a is subject to α-renaming. With that and quantifier mobility as stated above we in fact obtain a constraint entailing that x and y are different:

$$\exists a(x \doteq a) \wedge \exists a(y \doteq a) \;\equiv\; \exists a(x \doteq a) \wedge \exists b(y \doteq b) \;\equiv\; \exists a \exists b(x \doteq a \wedge y \doteq b).$$

Note that composition is not idempotent. Hence the expressions $\exists a(x \doteq a)$ and $\exists a(x \doteq a) \wedge \exists a(x \doteq a) \equiv \bot$ are not congruent.

3 Records

The constraint system underlying Oz [18] provides a domain of so-called feature trees that is closed under record construction. Since records are the congenial data structure for modelling object-oriented programming, we outline their constraint theory as far as is needed for the purposes of this paper. We will be very liberal as it comes to syntax. The reader may consult [18] for details.

Records are obtained with respect to an alphabet of constant symbols, called *atoms*, and denoted by a, b, f, g. Records are constructed (and possibly decomposed) by constraints of the form

$$x \doteq f(a_1{:}\, x_1 \ldots a_n{:}\, x_n)$$

where f is the *label*, $a_1, \ldots, a_n$ are the pairwise distinct field names, and $x_1, \ldots, x_n$ are the values of the record x. The order of the fields is not significant. A zero-field record $f()$ is identified with the atom f. The semantics of record construction is defined by the two axiom schemes

$$\begin{array}{rcll} f(\overline{a}{:}\,\overline{x}) \doteq f(\overline{a}{:}\,\overline{y}) & \leftrightarrow & \overline{x} \doteq \overline{y} & \\ f(\overline{a}{:}\,\overline{x}) \doteq g(\overline{b}{:}\,\overline{y}) & \rightarrow & \bot & \text{if } f \neq g \text{ or } [\overline{a}] \neq [\overline{b}] \end{array}$$

where $[\overline{a}]$ is the set of elements of the sequence $\overline{a}$. Field selection $x.y$ is a partial function on records satisfying the axiom schemes

$$\begin{array}{rcll} f(\overline{a}{:}\,\overline{x}\; b{:}\, y)\,.\,b & \doteq & y & \\ f(\overline{a}{:}\,\overline{x})\,.\,b \doteq y & \rightarrow & \bot & \text{if } b \notin [\overline{a}]. \end{array}$$

The function $\mathsf{label}(x)$ yields the label of a record according to the scheme

$$\mathsf{label}(f(\cdots)) \;\doteq\; f.$$

Finally, record adjunction "$\mathsf{adjoinAt}(x, y, z)$" adjoins a field $y{:}\, z$ to a record x:

$$\begin{array}{rcll} \mathsf{adjoinAt}(f(\overline{a}{:}\,\overline{x}\; b{:}\, y), b, z) & \doteq & f(\overline{a}{:}\,\overline{x}\; b{:}\, z) & \\ \mathsf{adjoinAt}(f(\overline{a}{:}\,\overline{x}), b, z) & \doteq & f(\overline{a}{:}\,\overline{x}\; b{:}\, z) & \text{if } b \notin [\overline{a}]\,. \end{array}$$

We write $f(x_1 \ldots x_n)$ as a short hand for the record $f(1{:}\,x_1 \ldots n{:}\,x_n)$. Thus we obtain Prolog terms as a special case of records. The outlined constraint system is in fact a conservative extension of Prolog II's rational tree system.

4 The Programming Language

Having glimpsed at the mathematical model of Oz, we are now ready to see the concrete programming language.

A procedure P taking n arguments can be defined with the concrete syntax

$$\textbf{proc}\ \{P\ X_1 \ldots X_n\}\ \ \sigma\ \ \textbf{end}$$

standing for the abstract expression

$$P{:}\, X_1 \ldots X_n/\sigma\ \wedge\ \exists a\,(P \doteq procedure(name{:}\, a\ arity{:}\, n)).$$

Thus a procedure definition introduces an abstraction and equips it with a unique identity. This construction ensures that a variable can link at most one abstraction on a consistent blackboard. Since the variable P denotes the record $procedure(name{:}\,a\ arity{:}\,n)$ rather than the abstraction, we can test for equality between P and other variables. The resulting first-class equality for procedures (i.e., procedure identities) provides for useful programming techniques. The fact that procedures have unique identity is also important for the efficient implementation of the reduction rule for applications.

The following expression defines a map function for lists in concrete Oz syntax:

```
proc {Map X P Y}
   if H T in
      X = H|T
   then
      Y = {P H}|{Map T P}
   else
      X = nil
      Y = nil
   fi
end
```

The atom *nil* stands for the empty list, and $H|T$ abbreviates the record $cons(H\ T)$ representing the list whose head is H and whose tail is T. The "$H\ T$ **in**" prefix quantifies the variables H and T in both the guard and the body of the clause. Composition is written as juxtaposition. Variables start with an upper-case letter and are thus distinguished from atoms, which start with a lower-case letter. The line $Y = \{P\ H\}|\{Map\ T\ P\}$ contains two nested applications, which are eliminated using auxiliary variables and composition:

$$\exists U\ \exists V\ (Y \doteq cons(U\ V)\ \wedge\ P\ H\ U\ \wedge\ Map\ T\ P\ V).$$

```
proc {Producer}
    exists Ack in
    item('yellow brick' Ack ok) ! Channel
    if Ack = ok then {Producer} fi
end
proc {Consumer}
    exists X Ack in
    item(X ok Ack) ? Channel
    if Ack = ok then {AddToRoad X} {Consumer} fi
end
```

Figure 3: Synchronized producer-consumer communication

We will use nested notation frequently, thus alleviating the verboseness of the purely relational calculus. (The Oz Calculus is designed purely relational since this setup provides for the minimal and orthogonal organization of its constructs; for instance, constraints are completely separated from the other constructs.)

Constraint communication is asynchronous. Synchronous communication can be expressed by combining constraint communication with the conditional. In the producer-consumer example in Figure 3, computation suspends until communication has taken place (signaled by an acknowledgement). The default for a missing else part of a conditional is **else true**. The nested get token *item(X ok Ack) ? Channel* translates into

$$\exists Y\,(Y \doteq item(X\ ok\ Ack)\ \wedge\ Y?Channel).$$

5 Objects

An object is a persistent agent processing messages from the outside world. It has a static aspect, its method table, and a dynamic aspect, its state. Although their state may change, objects do have a persistent identity. Methods are possibly indeterministic functions

$$method\colon state \times message \rightarrow state$$

defining the behavior of objects. Messages are processed as follows: First obtain the method name from the message, then obtain the corresponding method from the object's method table, and finally change to a possibly new state by applying the method to the current state and the message.

There are several possibilities for expressing objects in Oz. The one we will present here represents an object O as a procedure "sending" the message given as its argument.

```
proc {O Message Continuation}
   if Method in
      Method = MethodTable.{Label Message}
   then exists State in
      State ? Channel
      if {Label State} = state
      then {Method State Message} ! Channel
            {Continuation}
      fi
   fi
end
```

A message is represented as a record whose label is taken as the name of the requested method. The method table is represented as a record whose field names act as method names. The state of the object resides on the blackboard as a put token *State*!*Channel*, where only the object *O* is supposed to know the link *Channel*. The state is represented as a record whose fields act as the attributes of the object. The guard {*Label State*} = *state* suspends the application of the method until the state is known on the blackboard.

The argument *Continuation* of the procedure *O* is is a zero-argument procedure to be applied concurrently with the method. It provides for synchronization upon and sequentialization of message sending.

There is sugared syntax for message sending (**local** is a variant of **exists** having a closing **end**):

$$\text{O\^{}M};\ \sigma \implies \textbf{local}\ \text{P}\ \textbf{in}\ \{\text{O M P}\}\ \textbf{proc}\ \{\text{P}\}\ \sigma\ \textbf{end}\ \textbf{end}.$$

Moreover, *O^M* abbreviates *O^M*; **true**. Thus *O^M*; *O^N* sends first message *M* and then message *N* to the object *O*. Since we are in a concurrent setting, it is possible that *O* takes other messages between *M* and *N*.

Since objects are represented as procedures, they enjoy in fact persistent identity (recall the translation of **proc** ··· **end** given in Section 4). Thus one can test for identity of two objects *O1*, *O2* using a conditional **if** *O1* = *O2* **then** ··· **fi**. Note that many agents may know an object *O* and thus may concurrently attempt sending messages. Handling the state with constraint communication ensures mutual exclusion: the respective method applications are implicitly and indeterministically sequentialized since there will be at most one put token holding the state on the blackboard.

Since procedures are first-class citizens, we can write a generic procedure creating a new object from a method table and an initializing message:

```
proc {Create MethodTable IMessage O}
   exists Channel in
      {MethodTable.{Label IMessage} state(self:O) IMessage} ! Channel
      proc {O Message Continuation} ··· end
end
```

```
local Set Inc See in
    Counter = {Create mt(set:Set  inc:Inc  see:See) set(0)}
    proc {Set InState Message OutState}
        OutState = {AdjoinAt InState val Message.1}
    end
    proc {Inc InState Message OutState}
        OutState = {AdjoinAt InState val (InState.val + 1)}
    end
    proc {See InState Message OutState}
        OutState = InState
        Message.1 = InState.val
    end
end
```

Figure 4: A counter object in plain syntax

The notion of "self" is captured straightforwardly by equipping the initial state *state(self: O)* with a self-reference. Note that the object's state is encapsulated since quantification ensures that only the procedure *O* knows the link *Channel*.

To summarize, we are now in a position where we can create a concurrent object by simply applying the procedure *Create* to a method table and an initializing message. The method table may be seen as the class of the object. Both the object and its class are first-class citizens having unique identity. A message is sent by simply applying the object to it.

5.1 A Counter Object

Figure 4 shows how a counter can be set up as an object having methods for initializing, incrementing and reading its value. The initializing message *set(0)* adjoins the new attribute *val: 0* to the initial state *state(self: Counter)*. In fact, due to the semantics of *AdjoinAt* (see Section 3), every method may adjoin new attributes to an object's state.

Reduction of *Counter^see(X)* constrains the variable *X* to the current value of *Counter*. Reduction of

Counter^set(X); Counter^inc; Counter^inc; Counter^see(5)

constrains the variable *X* to 3, provided no one else is sending intervening messages to *Counter*. We will see in the next section how this can be prevented. The above reduction illustrates the smooth integration of the notions of state and logic variable.

Oz supports special syntax for object creation and method definition, which allows writing the expression in Figure 4 as follows:

```
create Counter with set(0)
    meth set(X)   val ← X end
    meth inc      val ← @val + 1 end
    meth see(X)   X = @val end
end
```

5.2 Inheritance

The behavior of an object is determined by its method table. Inheritance thus means that the method table of a new object is obtained by combining and extending method tables of existing objects. Since method tables are represented as first-class values, combining and extending them is straightforward (e.g., by record adjunction). To make the methods of an object accessible, we will now represent an object as a record

object(table: MT send: Q)

where *MT* is the method table and *Q* is the previous object representation. The sugared syntax for synchronized message sending translates now as follows:

$$\mathrm{O\hat{\ }M};\ \sigma \implies \textbf{local}\ \mathrm{P}\ \textbf{in}\ \{\mathrm{O.send\ M\ P}\}\ \textbf{proc}\ \{\mathrm{P}\}\ \sigma\ \textbf{end end}.$$

With the new object representation we can create an object *DecCounter* by inheriting the methods of *Counter* and adding a method for decrementing the value:

```
local Dec in
   DecCounter = {Create {AdjoinAt Counter.table dec Dec} set(0)}
   proc {Dec InState Message OutState}
      OutState = {AdjoinAt InState val (InState.val - 1)}
   end
end
```

In sugared syntax we can write more nicely:

```
create DecCounter from Counter with set(0)
   meth dec  val ← @val - 1 end
end
```

To create a new counter *C* having exactly the same methods as *DecCounter* and taking *X* as initial value, we simply write

```
create C from DecCounter with set(X) end.
```

Observe that our model alleviates the distinction between classes and their instances by combining object creation and inheritance into one single operation.

In a concurrent setting it is sometimes essential to send an object a block of messages to be processed without intervening messages. The ability to obtain and release locks on objects is equally important. To this purpose we define an object with a single method *batch* taking a list of messages as argument [10]:

```
create BatchObject with batch(nil)
    meth batch(L)
        if H T in L=H|T then ⟨⟨@self H⟩⟩ ⟨⟨@self batch(T)⟩⟩ fi
    end
end
```

The two consecutive *message applications* are threaded with an intermediate state

$$\exists State\ ({}^{InState}\langle\langle @self\ H\rangle\rangle^{State} \wedge {}^{State}\langle\langle @self\ batch(T)\rangle\rangle^{OutState})$$

and a threaded message application ${}^{InState}\langle\langle O\ Message\rangle\rangle^{OutState}$ expands into

{O.table.{Label Message} InState Message OutState}.

The notation for message application exploits the fact that in our model every method m of every object O can be referred to by $O.table.m$. Incidentally, our notation for message application also serves the purpose of Smalltalk's "super" notation.

A decrementable counter with a batch method can now be obtained by multiple inheritance from *DecCounter* and *BatchObject*:

create C **from** DecCounter BatchObject **with** set(0) **end**

The method table of C is obtained by adjoining the tables of *DecCounter* and *BatchObject*. Now

C^batch(set(X) | inc | inc | see(5) | nil)

is guaranteed to constrain X to 3 (compare with the example in Section 5.1).

6 Summary

Oz is an attempt to create a high-level concurrent programming language bringing together the merits of logic and object-oriented programming. For this purpose, we extend the concurrent constraint model with a facility for higher-order programming and the new notion of constraint communication. The semantics of Oz is specified by a new mathematical model, called the Oz Calculus. In addition to higher-order programming and constraint communication, the Oz Calculus provides an abstract compositional semantics for deep guards and the dynamic creation of new and unique names.

We have shown how concurrent objects created by multiple inheritance can be expressed concisely and naturally in Oz. Objects, classes, methods and messages are all modeled as first-class citizens. Although objects change their state, they enjoy persistent identity. The object model profits from the fact that the constraint system underlying Oz provides records as logic data structure.

An implementation of Oz based on a compiler and an abstract machine written in C++ exists and shows encouraging performance. Efficient implementation of constraint communication is not difficult. The construction of new states by record adjunction can be safely optimized to destructive assignment (i.e., compile-time garbage collection) if the compiler enforces certain syntactic restrictions.

Acknowledgements

We thank all members of the Programming Systems Lab at DFKI for inspiring discussions on all kinds of subjects and objects; particularly many suggestions came from Michael Mehl, Ralf Scheidhauer, and Ralf Treinen. The research reported in this paper has been supported by the Bundesminister für Forschung und Technologie, contract ITW 9105 (Hydra), and by the ESPRIT basic research project 7195 (ACCLAIM).

References

[1] H. Aït-Kaci, A. Podelski, and G. Smolka. A feature-based constraint system for logic programming with entailment. In FGCS'92 [6], pages 1012–1021.

[2] W. Chen, M. Kifer, and D. S. Warren. Hilog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15:187–230, 1993.

[3] K. Clark and S. Gregory. A relational language for parallel programming. In *Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 171–178, 1981.

[4] A. Colmerauer and F. Benhamou, editors. *Constraint Logic Programming: Selected Research*. The MIT Press, Cambridge, Mass., 1993. To appear.

[5] A. Colmerauer, H. Kanoui, and M. V. Caneghem. Prolog, theoretical principles and current trends. *Technology and Science of Informatics*, 2(4):255–292, 1983.

[6] *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, 1992. ICOT.

[7] Y. Goldberg, W. Silverman, and E. Shapiro. Logic programs with inheritance. In FGCS'92 [6], pages 951–960.

[8] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, Jan. 1987.

[9] S. Janson and S. Haridi. Programming paradigms of the Andorra kernel language. In V. Saraswat and K. Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186, San Diego, USA, 1991. The MIT Press.

[10] K. Kahn. Objects: A fresh look. In *Proceedings of the Third European Conference on Object Oriented Programming*, pages 207–223. Cambridge University Press, Cambridge, MA, 1989.

[11] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 858–876, Cambridge, MA, 1987. The MIT Press.

[12] 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, Oct. 1991.

[13] G. Nadathur and D. Miller. An overview of λProlog. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 810–827, Seattle, Wash., 1988. The MIT Press.

[14] V. 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.

[15] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–511, Sept. 1989.

[16] E. Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing*, 1:24–48, 1983.

[17] G. Smolka. A calculus for higher-order concurrent constraint programming. Research report, DFKI, Postfach 2080, 6750 Kaiserslautern, Germany, 1993. Forthcoming.

[18] G. Smolka and R. Treinen. Records for logic programming. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 240–254, Washington, USA, 1992. The MIT Press. Full version has appeared as Research Report RR-92-23, DFKI, Stuhlsatzenhausweg 3, 6600 Saarbrücken 11, Germany.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
D-6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all published papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-92-21
Jörg-Peter Mohren, Jürgen Müller
Representing Spatial Relations (Part II) -The Geometrical Approach
25 pages

RR-92-22
Jörg Würtz: Unifying Cycles
24 pages

RR-92-23
Gert Smolka, Ralf Treinen:
Records for Logic Programming
38 pages

RR-92-24
Gabriele Schmidt: Knowledge Acquisition from Text in a Complex Domain
20 pages

RR-92-25
Franz Schmalhofer, Ralf Bergmann, Otto Kühn, Gabriele Schmidt: Using integrated knowledge acquisition to prepare sophisticated expert plans for their re-use in novel situations
12 pages

RR-92-26
Franz Schmalhofer, Thomas Reinartz, Bidjan Tschaitschian: Intelligent documentation as a catalyst for developing cooperative knowledge-based systems
16 pages

RR-92-27
Franz Schmalhofer, Jörg Thoben: The model-based construction of a case-oriented expert system
18 pages

RR-92-29
Zhaohui Wu, Ansgar Bernardi, Christoph Klauck:
Skeletel Plans Reuse: A Restricted Conceptual Graph Classification Approach
13 pages

RR-92-30
Rolf Backofen, Gert Smolka:
A Complete and Recursive Feature Theory
32 pages

RR-92-31
Wolfgang Wahlster:
Automatic Design of Multimodal Presentations
17 pages

RR-92-33
Franz Baader: Unification Theory
22 pages

RR-92-34
Philipp Hanschke: Terminological Reasoning and Partial Inductive Definitions
23 pages

RR-92-35
Manfred Meyer:
Using Hierarchical Constraint Satisfaction for Lathe-Tool Selection in a CIM Environment
18 pages

RR-92-36
Franz Baader, Philipp Hanschke:
Extensions of Concept Languages for a Mechanical Engineering Application
15 pages

RR-92-37
Philipp Hanschke: Specifying Role Interaction in Concept Languages
26 pages

RR-92-38
Philipp Hanschke, Manfred Meyer:
An Alternative to H-Subsumption Based on Terminological Reasoning
9 pages

RR-92-40
Philipp Hanschke, Knut Hinkelmann: Combining Terminological and Rule-based Reasoning for Abstraction Processes
17 pages

RR-92-41
Andreas Lux: A Multi-Agent Approach towards Group Scheduling
32 pages

RR-92-42
John Nerbonne:
A Feature-Based Syntax/Semantics Interface
19 pages

RR-92-43
Christoph Klauck, Jakob Mauss: A Heuristic driven Parser for Attributed Node Labeled Graph Grammars and its Application to Feature Recognition in CIM
17 pages

RR-92-44
Thomas Rist, Elisabeth André: Incorporating Graphics Design and Realization into the Multimodal Presentation System WIP
15 pages

RR-92-45
Elisabeth André, Thomas Rist: The Design of Illustrated Documents as a Planning Task
21 pages

RR-92-46
Elisabeth André, Wolfgang Finkler, Winfried Graf, Thomas Rist, Anne Schauder, Wolfgang Wahlster: WIP: The Automatic Synthesis of Multimodal Presentations
19 pages

RR-92-47
Frank Bomarius: A Multi-Agent Approach towards Modeling Urban Traffic Scenarios
24 pages

RR-92-48
Bernhard Nebel, Jana Koehler:
Plan Modifications versus Plan Generation: A Complexity-Theoretic Perspective
15 pages

RR-92-49
Christoph Klauck, Ralf Legleitner, Ansgar Bernardi: Heuristic Classification for Automated CAPP
15 pages

RR-92-50
Stephan Busemann:
Generierung natürlicher Sprache
61 Seiten

RR-92-51
Hans-Jürgen Bürckert, Werner Nutt:
On Abduction and Answer Generation through Constrained Resolution
20 pages

RR-92-52
Mathias Bauer, Susanne Biundo, Dietmar Dengler, Jana Koehler, Gabriele Paul: PHI - A Logic-Based Tool for Intelligent Help Systems
14 pages

RR-92-54
Harold Boley: A Direkt Semantic Characterization of RELFUN
30 pages

RR-92-55
John Nerbonne, Joachim Laubsch, Abdel Kader Diagne, Stephan Oepen: Natural Language Semantics and Compiler Technology
17 pages

RR-92-56
Armin Laux: Integrating a Modal Logic of Knowledge into Terminological Logics
34 pages

RR-92-58
Franz Baader, Bernhard Hollunder:
How to Prefer More Specific Defaults in Terminological Default Logic
31 pages

RR-92-59
Karl Schlechta and David Makinson: On Principles and Problems of Defeasible Inheritance
13 pages

RR-92-60
Karl Schlechta: Defaults, Preorder Semantics and Circumscription
19 pages

RR-93-02
Wolfgang Wahlster, Elisabeth André, Wolfgang Finkler, Hans-Jürgen Profitlich, Thomas Rist:
Plan-based Integration of Natural Language and Graphics Generation
50 pages

RR-93-03
Franz Baader, Berhard Hollunder, Bernhard Nebel, Hans-Jürgen Profitlich, Enrico Franconi:
An Empirical Analysis of Optimization Techniques for Terminological Representation Systems
28 pages

RR-93-04
Christoph Klauck, Johannes Schwagereit:
GGD: Graph Grammar Developer for features in CAD/CAM
13 pages

RR-93-05
Franz Baader, Klaus Schulz: Combination Techniques and Decision Problems for Disunification
29 pages

RR-93-06
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux: On Skolemization in Constrained Logics
40 pages

RR-93-07
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux: Concept Logics with Function Symbols
36 pages

RR-93-08
Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer: COLAB: A Hybrid Knowledge Representation and Compilation Laboratory
64 pages

RR-93-09
Philipp Hanschke, Jörg Würtz:
Satisfiability of the Smallest Binary Program
8 Seiten

RR-93-10
Martin Buchheit, Francesco M. Donini, Andrea Schaerf: Decidable Reasoning in Terminological Knowledge Representation Systems
35 pages

RR-93-11
Bernhard Nebel, Hans-Juergen Buerckert:
Reasoning about Temporal Relations:
A Maximal Tractable Subclass of Allen's Interval Algebra
28 pages

RR-93-12
Pierre Sablayrolles: A Two-Level Semantics for French Expressions of Motion
51 pages

RR-93-13
Franz Baader, Karl Schlechta:
A Semantics for Open Normal Defaults via a Modified Preferential Approach
25 pages

RR-93-14
Joachim Niehren, Andreas Podelski, Ralf Treinen: Equational and Membership Constraints for Infinite Trees
33 pages

RR-93-15
Frank Berger, Thomas Fehrle, Kristof Klöckner, Volker Schölles, Markus A. Thies, Wolfgang Wahlster: PLUS - Plan-based User Support Final Project Report
33 pages

RR-93-16
Gert Smolka, Martin Henz, Jörg Würtz: Object-Oriented Concurrent Constraint Programming in Oz
17 pages

DFKI Technical Memos

TM-91-12
Klaus Becker, Christoph Klauck, Johannes Schwagereit: FEAT-PATR: Eine Erweiterung des D-PATR zur Feature-Erkennung in CAD/CAM
33 Seiten

TM-91-13
Knut Hinkelmann: Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

TM-91-14
Rainer Bleisinger, Rainer Hoch, Andreas Dengel: ODA-based modeling for document analysis
14 pages

TM-91-15
Stefan Busemann: Prototypical Concept Formation
An Alternative Approach to Knowledge Representation
28 pages

TM-92-01
Lijuan Zhang: Entwurf und Implementierung eines Compilers zur Transformation von Werkstückrepräsentationen
34 Seiten

TM-92-02
Achim Schupeta: Organizing Communication and Introspection in a Multi-Agent Blocksworld
32 pages

TM-92-03
Mona Singh:
A Cognitiv Analysis of Event Structure
21 pages

TM-92-04
Jürgen Müller, Jörg Müller, Markus Pischel, Ralf Scheidhauer:
On the Representation of Temporal Knowledge
61 pages

TM-92-05
Franz Schmalhofer, Christoph Globig, Jörg Thoben: The refitting of plans by a human expert
10 pages

TM-92-06
Otto Kühn, Franz Schmalhofer: Hierarchical skeletal plan refinement: Task- and inference structures
14 pages

TM-92-08
Anne Kilger: Realization of Tree Adjoining Grammars with Unification
27 pages

TM-93-01
Otto Kühn, Andreas Birk: Reconstructive Integrated Explanation of Lathe Production Plans
20 pages

DFKI Documents

D-92-11
Kerstin Becker: Möglichkeiten der Wissensmodellierung für technische Diagnose-Expertensysteme
92 Seiten

D-92-12
Otto Kühn, Franz Schmalhofer, Gabriele Schmidt: Integrated Knowledge Acquisition for Lathe Production Planning: a Picture Gallery (Integrierte Wissensakquisition zur Fertigungsplanung für Drehteile: eine Bildergalerie)
27 pages

D-92-13
Holger Peine: An Investigation of the Applicability of Terminological Reasoning to Application-Independent Software-Analysis
55 pages

D-92-14
Johannes Schwagereit: Integration von Graph-Grammatiken und Taxonomien zur Repräsentation von Features in CIM
98 Seiten

D-92-15
DFKI Wissenschaftlich-Technischer Jahresbericht 1991
130 Seiten

D-92-16
Judith Engelkamp (Hrsg.): Verzeichnis von Softwarekomponenten für natürlichsprachliche Systeme
189 Seiten

D-92-17
Elisabeth André, Robin Cohen, Winfried Graf, Bob Kass, Cécile Paris, Wolfgang Wahlster (Eds.): UM92: Third International Workshop on User Modeling, Proceedings
254 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-92-18
Klaus Becker: Verfahren der automatisierten Diagnose technischer Systeme
109 Seiten

D-92-19
Stefan Dittrich, Rainer Hoch: Automatische, Deskriptor-basierte Unterstützung der Dokumentanalyse zur Fokussierung und Klassifizierung von Geschäftsbriefen
107 Seiten

D-92-21
Anne Schauder: Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars
57 pages

D-92-22
Werner Stein: Indexing Principles for Relational Languages Applied to PROLOG Code Generation
80 pages

D-92-23
Michael Herfert: Parsen und Generieren der Prolog-artigen Syntax von RELFUN
51 Seiten

D-92-24
Jürgen Müller, Donald Steiner (Hrsg.): Kooperierende Agenten
78 Seiten

D-92-25
Martin Buchheit: Klassische Kommunikations- und Koordinationsmodelle
31 Seiten

D-92-26
Enno Tolzmann: Realisierung eines Werkzeugauswahlmoduls mit Hilfe des Constraint-Systems CONTAX
28 Seiten

D-92-27
Martin Harm, Knut Hinkelmann, Thomas Labisch: Integrating Top-down and Bottom-up Reasoning in COLAB
40 pages

D-92-28
Klaus-Peter Gores, Rainer Bleisinger: Ein Modell zur Repräsentation von Nachrichtentypen
56 Seiten

D-93-01
Philipp Hanschke, Thom Frühwirth: Terminological Reasoning with Constraint Handling Rules
12 pages

D-93-02
Gabriele Schmidt, Frank Peters, Gernod Laufkötter: User Manual of COKAM+
23 pages

D-93-03
Stephan Busemann, Karin Harbusch(Eds.): DFKI Workshop on Natural Language Systems: Reusability and Modularity - Proceedings
74 pages

D-93-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1992
194 Seiten

D-93-06
Jürgen Müller (Hrsg.): Beiträge zum Gründungsworkshop der Fachgruppe Verteilte Künstliche Intelligenz Saarbrücken 29.-30. April 1993
235 Seiten
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

Object-Oriented Concurrent Constraint Programming in Oz

Gert Smolka, Martin Henz, Jörg Würtz

RR-93-16

Research Report