Concurrent Constraint Programming: A brief survey

Vijay Saraswat Xerox PARC
3333 Coyote Hill Road
Palo Alto Ca 94304
(saraswat@parc.xerox.com)

August 1992

Since I submitted my thesis [Sar89], there has been considerable progress on different aspects of concurrent constraint programming. What follows is a brief and somewhat personal review of some of the progress; it is based on a chapter I added as an epilog to [Sar89]. In most cases I will try to convey the intuitions underlying the work, rather than discuss the formal results.

1 Constraint systems

A simple definition of an abstract notion of a constraint system has been surprisingly hard to come by. One would like a definition that captures the generality of the notion while still being tractable enough to allow a simple development of the formal semantics.

The basic notion is simple enough. A constraint system is a system of partial information, as in the treatment of [Sco82]. However, there is some additional logical structure involved, namely that of free variables, existential quantification and substitution. Indeed, one may well think of constraint systems as being the *first-order* analogs of the (propositional) information systems. The question is how best to introduce the extra logical structure.

We initially experimented with the "algebraic" treatment of logic [SRP91] — the cylindric algebras of Henkin, Monk and Tarski, and the polyadic algebras of Halmos. The first approach introduces an underlying set `Var` of variables, a formal operation of existential quantification indexed by each variable in `Var`, and *diagonal* elements, indexed by pairs of elements in `Var`. Various axioms are given specifying the properties of these operations, and governing their interactions. Halmos' approach is similar, except that he takes as basic existential quantifiers and substitutions, rather than existential quantifiers and the equality relation.

The natural evolution of this approach is to examine the very elegant development of *hyperdoctrines* in the categorical approach to logic founded by Lawvere. Here it is possible to do away with variables completely, and instead get the effect of substitution through the composition of arrows. Essentially, a constraint system can be treated as a functor between a "base" category, and a category of meet-preorders (reflecting the entailment relation connecting tokens).

A much more elementary route is through proof-theory [Sar92]. Here we take a constraint system to be a collection of formulas closed under conjunction and existential quantification, and satisfying (a) the (left and right introduction) proof rules for (the sequent-style presentation of) the $(\wedge, \exists)$-fragment of intuitionistic logic, and (b) a certain "genericity" condition, which essentially states that free variables on both sides of $\vdash$ are universally quantified.

Of course, the usual constraint systems defined by means of entailment in first-order theories satisfy such a formal definition. In addition, it is possible to carry out the usual development of information systems in this richer setting, and establish that constraint systems form a category, the category has products and exponentiation (function-space), and that the two mesh together as is required for the category to be Cartesian-closed. This means that "recursive" domain equations can be solved over constraint systems, and opens one door to the development of higher-order constraint languages.

The full potential of these ideas for the systematic construction of constraint systems is in urgent need of exploration.

2 Model-theory of cc languages.

2.1 Determinate cc languages

Eighteen years ago, Kahn gave a theory of determinate data-flow [Kah74]. Today this theory (with some extensions, e.g. to concrete data structures) remains the dominant theory of determinate concurrent computation. While simple, elegant and widely applicable, the theory's emphasis on computing with directional point-to-point communication channels imposes some expressibility restrictions: channels carry only fully formed values from the underlying data domain; networks are not dynamically reconfigurable in that connections cannot themselves be passed as data on communication channels; and each channel has only one reader and one writer. Because of these restrictions, Kahn's theory cannot model such useful programming idioms as incomplete messages, difference lists, recursive doubling, multiple readers/writers of a communication channel, dynamic reconfiguration, etc. Although these techniques were originally developed in the context of indeterminate concurrent logic programming, they are all both inherently determinate and usefully expressible in determinate contexts.

The notion of *determinate concurrent constraint programming* languages was explicitly isolated in [SRP91]. The combinators treated are Ask, Eventual Tell, parallel composition, hiding and recursion. A simple model was presented, based on the specification-oriented methodology of [OH86]. The crucial insight was to focus on observing the *resting points* of a process—those stores in which the process quiesces without producing more information. It turns out that for the determinate language, the set of resting points of a process *completely characterizes* its behavior on all inputs, since each process can be identified with a closure operator over the underlying constraint system. Very natural definitions of parallel composition, communication and hiding are given. For example, the parallel composition of two agents can be characterized by just the intersection of the sets of constraints associated with them. We also gave a complete axiomatization of equality in this model, and showed that the model is fully abstract with respect to this semantics.

Closure operators. The discovery of the connection between closure operators and concurrent constraint programming was a fundamental one.[1] Closure operators are functions f over a partial order L that are extensive $f(x) \geq x$, monotone ($f(x) \geq f(y)$ whenever $x \geq y$) and idempotent ($f(f(x)) = x$). Idempotence says that the range of f is equal to the set of its fixed-points; extensiveness says that each point in the domain is mapped to a fixed-point above it, and monotonicity ensures that this must be the *least* such fixed point. Hence, a closure operator can be uniquely represented by the set of its fixed-points (its range).

Closure operators capture precisely the idea of monotonic accumulation of information inherent in the cc conception. On the one hand this discovery showed us that the mathematical objects underlying (at least the determinate) cc languages were very simple indeed, and had many interesting properties. (For example, the set of resting points of the process $f \wedge g$ is the intersection of the resting points of f and g — from this it is immediate that parallel composition is commutative, associative, idempotent etc.) On the other, by thinking of the cc languages as *the* language framework for computing with closure operators we were lead to a number of simple extensions with interesting computational content.

Determinate Disjunction. The simplest example is the *glb* operation. The set of closure operators forms a lattice, with the least upper bound given by parallel composition. What does the greatest lower bound of f and g give us? On any input c, it gives us the greatest lower bound (in the space of constraints)

[1] For this, I am indebted to Keshav Pingali and Tony Hoare, who had been using these operators in their own work. Together with Radha Jagadeesan and Prakash Panangaden, Keshav had developed these ideas in the context of a first-order functional programming language with logic variables [JPP89]; a language that can be seen as an instance of Determinate CCP. Tony Hoare had been developing a theory of "conjunction and concurrency" [Hoa89] which is an aspect of Detrminate CCP.

of $f(c)$ and $g(c)$ — one can think of it is a *determinate or*: it extracts the most *common* information between the two branches. [HSD92] shows how this can be used to obtain considerable improvement in some constraint propagation algorithms.

Other examples of operations on constraints (e.g., indexical constraints, cardinality operators) that can readily be understood in terms of closure operators may be found in [HSD92].

Categorial Treatment. A word about the categorial semantics of (determinate) concurrent constraint programming. One can think of the underlying constraint language as providing the structure of existential quantification and conjunction on first-order systems of partial information. Determinate CCP, then, "lifts" this structure up to the level of computational processes: existential quantification translates to hiding, conjunction translates to parallel composition. Is there some way in which this can be made precise?

[PSSS91] does this in the setting of a categorial approach to CCP. As mentioned earlier, constraint systems can be thought of as certain kinds of hyperdoctrines. The paper shows how to build another hyperdoctrine out of closure operators, as a functorial image of the hyperdoctrine describing constraint systems. The functor maps conjunction and existential quantification to the corresponding maps on closure operators that interpret parallel composition and hiding for processes; it thus describes precisely how the logical structure at the underlying level is lifted up to structure at the level of agents.

2.2 Nondeterminate cc language

It is natural to ask what happens when disjunction is added to the picture. This thesis shows one way of doing this operationally — one adds a notion of non-determinate prefixing ($\Rightarrow$), and takes as configurations multisets of pairs of agents and constraints.

However, it is not clear how to connect non-deterministic prefixing to logical notions. Instead [JSS91] explores another route. Let $(A_1 \vee A_2)$ be an agent if A_1 and A_2 are. Operationally, as with nondeterminate prefixing, an agent $(A_1 \vee A_2)$ in store c splits into *two* configurations, $\langle A_1, c\rangle$ and $\langle A_2, c\rangle$. (Conjunctively running agents of $(A_1 \vee A_2)$ are copied into both.) When the store of one branch becomes inconsistent, the branch vanishes. The crucial question now is: what does one regard as *observable* for such a system of disjunctive agents?

One answer is that the observables of such an agent should be the multiset of constraints obtained on each branch. However, this conflicts with the operational requirement that "false" branches are unobservable: an agent which produces false on one branch and `X=1` on another should not be distinguished from the agent that just produces `X=1`.

This leads to the notion that two agents are considered operationally indistinguishable if any contraint c entailed by *each* branch of one is also entailed by each branch of the other, and vice versa. Indeed, this corresponds to the reading of $\vee$ as disjunction: $A \vee B$ entails c just in case A entails c and B entails c.

[JSS91] develops the mathematical foundation of such languages. As in the determinate case, it turns out to be adequate to record for a process its *resting points*: those stores in which the process is unable to add any more information (on at least *one* of its branches). Again, the underlying mathematical reason is that processes can be taken to denote certain kinds of closure operators, namely *non-deterministic*, *linear* closure operators.[2] Such operators can also be represented by the set of their fixed-points; but now one is not guaranteed that there is a *minimal* fixed point above every input. Rather the result of the computation is taken to be the set of minimal elements above the input.

This line of development is still unfinished however. Clearly, other treatments of disjunction are possible, and should be investigated.

2.3 Indeterminate cc languages

The emphasis on recording for a process its resting points pays perhaps the richest dividends in the treatment of indeterminate ("committed-choice") cc languages [SRP91].

The denotation of an indeterminate process is only slightly more complex than in the determinate case: rather than storing just the resting points of a process, we must also associate with each resting point the "path" followed by the process in reaching that point. Such a path is also called a *failure*. One of the primary contributions of [SRP91] was to develop a simple treatment of failures via the notion of trace operators (see below).

However, in a fashion well-known from work on the semantics of other indeterminate formalisms, there are still a variety of choices for the appropriate mathematical domain of processes, depending on what one wants to consider observable.

The simplest choice, perhaps, is to observe for a process just the set of all stores that it can lead to, that is, stores it "deadlocks" in (that is, is unable to progress in, unless the environment intervenes to upgrade the store). Such a semantics would be useful, for example, in proving safety (or "partial correctness") properties for the programs: one would be able to show that regardless of the store the process ends up in, some given property holds. One can obtain a fully abstract semantics for this notion of observation by associating with a process just the set of all failures (resting points, together with the paths taken to reach that point) generable by the process. Of importance is the fact that

[2] *Non-deterministic* because the operators are functions over a powerdomain (the Smyth powerdomain), since a process is capable of producing more than one result from a single store. *Linear* because the operators do not share any information between stores: the set of (minimal) results that a process can produce from the set of stores $s \cup t$ is the union of the results it can produce on s and t.

the divergent or undefined process (needed for the base case of the semantic recursion defining the semantics of recursive processes) is just the empty set. This is because there is no store that one can observe such a process quiescing in. Indeed, a consequence of this notion of observation is that a process $\mathtt{c} \rightarrow \mathtt{A}$ is identified with the process $(\mathtt{c} \rightarrow \mathtt{div}) \Box (\mathtt{c} \rightarrow \mathtt{A})$, which can non-deterministically choose to diverge. (Here `div` is just some process that can engage in an infinite execution sequence without producing any constraint or demanding any constraint from its environment.)

A minor modification of this treatment is to observe for a process the set of stores it quiesces in, together with a bit indicating whether this store was *terminal*, that is, the process had reached a stage where it would be unable to add any more information even if the environment were to upgrade the information content of the store. For example, `X=1` is a terminal resting point of the agent `X=1`, but not of the agent $(\mathtt{X} = \mathtt{1}, \mathtt{Y} = \mathtt{2}) \rightarrow \mathtt{Z} = \mathtt{3}$. With such a notion of observation we can distinguish between "deadlocked" programs (programs that have quiesced but can continue if the environment were to offer more information), and "terminated" programs. It is straightforward to develop a compositional semantics which is fully-abstract for this notion of observation.

The biggest drawback of the above treatment, however, is that it is not useful for reasoning about *liveness* properties of a program, that is for establishing that a process *must* be able to produce some constraints, regardless of the indeterminate choices that the underlying scheduler may make. For example, we know that the process

---

$$(\mathtt{X} = \mathtt{1} \rightarrow (\mathtt{Y} = \mathtt{1},\ \mathtt{Z} = \mathtt{2})) \Box (\mathtt{R} = \mathtt{1} \rightarrow (\mathtt{Y} = \mathtt{1},\ \mathtt{S} = \mathtt{2})).$$

---

must produce the constraint `Y=1` on any store stronger than either `X=1` or `R=1`.

The main difficulty is treating non-termination or *divergence*. This is handled in [SRP91] by regarding a divergent process as "catastrophic", in the style of [OH86,Hoa90]. Such a process is modelled as capable of exhibiting any failure whatsoever. Thus, the process $(\mathtt{c} \rightarrow \mathtt{div}) \Box (\mathtt{c} \rightarrow \mathtt{A})$ is identified with $\mathtt{c} \rightarrow \mathtt{div}$: a process that in store `c` can behave completely unpredictably (and hence of which no total correctness assertions can be made).

The set of all processes is identified by presenting some naturally motivated closure conditions on sets of failures, following [Jos90]. (For example, one of the major conditions ensures that processes are finitely nondeterminate.) The resulting notion of a process is intuitive and easy to understand. Combinators corresponding to ask, tell, nondeterminate (dependent) choice, parallel composition and hiding are defined. A simple operational semantics is given, and a full correspondence with the denotational semantics is established.

A very simple relationship was also established between the indeterminate and determinate semantics, showing that the two semantics are essentially identical for determinate programs over finitary constraint systems.

In summary, [SRP91] presents a simple model for the `cc` languages that is fully able to account for nondeterminism and divergence, thus permitting the use of this model in reasoning about liveness properties of programs. The model is also shown to be fully consistent with the intuitive operational semantics of `cc` programs.

Trace operators. A central issue in our semantic treatment is the representation of a failure. A failure could be represented as a sequence of ask/tell annotated constraints. Such a strategy is followed in earlier work on related languages, for example in [Sar85,Lev88,GL90], [GCLS88], [GMS89] etc. However, ask-tell sequences are far too concrete in this setting. They store too much information which must then be abstracted from (usually via complex closure conditions) since they encode the precise path followed by a process to arrive at a resting point. Furthermore, the definition of various combinators becomes rather cumbersome because of the need to "propagate" information down the trace. Instead, we chose to represent observations via a certain kind of closure operators. To capture various operational notions of interest, we introduce the concept of *trace operators*, and *bounded trace operators*.

Intuitively, a trace operator corresponds to the closure operators generated by "straight line" (non-branching) ask-and-tell `cc` programs. Mathematically, their characterizing property is that they are *invertible*: to every trace operator f there corresponds another trace operator f^{-1} such that the only resting point of $f \wedge f^{-1}$ is `false` (the top of the lattice). Intuitively, f^{-1} is obtained from f by taking the characteristic ask-and-tell sequence of f and "inverting" it, making every ask a tell, and every tell an ask. Thus, on any input, the conjunction of the two processes is able to tell exactly what the other asks, and thus proceed hand-over-hand, so to speak, all the way to `false`. Hence `false` is the only resting point for their conjunction. For instance, the inverse for the trace operator

$$(\mathtt{W} = 1) \rightarrow (\mathtt{X} = 2 \wedge (\mathtt{Y} = 3 \rightarrow \mathtt{Z} = 4))$$

is

$$(\mathtt{W} = 1) \wedge (\mathtt{X} = 2 \rightarrow (\mathtt{Y} = 3 \wedge (\mathtt{Z} = 4 \rightarrow \mathtt{false})))$$

A *bounded* trace operator is a trace operator that can stop short of `false`; that is, it is a trace operator on a sublattice obtained by considering all the elements below a certain element in the lattice.

"Assume/Tell" sequences. A related line of attack has been independently pursued by Frank De Boer, Catuscia Palamidessi and their colleagues [dBP91a, dBKPR90]. In [dBP90,dBP91b], the authors introduced the idea that it was possible to give a semantics to the asynchronous concurrent logic programming languages in a style familiar from the semantics of indeterminate data-flow languages. With a process were associated the set of its *assume-tell* sequences: a sequence $\mathtt{ask}(\mathtt{c}_1).\mathtt{tell}(\mathtt{c}_2).\mathtt{ask}(\mathtt{c}_3)\ldots\mathtt{tell}(\mathtt{c}_n)$ is associated with a process if when initiated in store $\mathtt{c}_1$, it can upgrade it to store $\mathtt{c}_2$, and then when initiated in $\mathtt{c}_3$ it can upgrade it to c_4 and so on, until when initiated in $\mathtt{c}_{n-1}$ it can upgrade it to c_n and then quiesce. The denotation of a process is then taken to a set of such sequences closed under some conditions which reflect the asynchronous nature of communication, and the lack of granularity of the Eventual Tell operation.[3]

The resulting model are very similar in nature to the trace-operator based models discussed above — indeed trace operators are just an abstract counterpart of such assume/tell sequences. (A significant technical difference is that the authors do not handle the interaction between divergence and non-determinism, so that their semantics is not adequate for dealing with liveness properties of processes.) The basic point made by the authors — that for asychronous languages the "branching structure" of the refusal set semantics of [BHR84] is not needed — is indeed valid, and is further corroborated by the work of [Jos90]. Indeed, deBoer, Palamidessi and their colleagues have gone on to demonstrate the generality of their approach by applying it to other asynchronous languages in which communication is not "extensive" [dBKPR90].

[dBP92] discusses an equational axiomatization of the congruence on cc processes induced by the above semantics. This is very valuable for it serves as the foundation on which to build a "work-bench" that can be used to construct and reason about programs. A similar axiomatization for the determinate language is provided in [SRP91].

Bisimulation semantics. [SR90] develops a different approach to the semantics of CCP. Following the operationally motivated work of Milner (and Park) [Mil80] (see [Mil90b] for an uptodate account), we introduced the notion of *reactive equivalence* of cc programs.

Let the "environment" in which an agent is executed be an abstraction of the rest of the computational system—including concurrently executing agents—with which the agent must interact to complete its computation. In order to interact, an agent must have certain variables—the *visible variables*—in common with its environment. A basic action is said to be a *visible action* if the agent's capability to engage in that action may be influenced by the environment. Thus a visible tell action is an action that imposes "new" constraints (i.e., constraints which are not yet known to have been imposed) on the visible variables, and a visible ask action is one which checks whether some new

[3] In a language without Atomic Tell operations, it is not possible to distinguish between the agent $\mathtt{tell}(c_1), \mathtt{tell}(c_2)$ and the agent $\mathtt{tell}(c)$, where c is logically equivalent to the conjunction of c_1 and c_2. Alternatively, each communication $\mathtt{tell}(c)$ can be thought of as happening in arbitrarily many, finite "pieces".

constraints have been imposed on the visible variables. Note that a visible action is not something that is "directly seen" by another agent—rather it is a step in the agent's own computation which could have been influenced by the action of some other agent. Such circumspection is necessary in understanding this concept because communication in cc languages is very indirect, and hence the concepts developed in the CCS/CSP framework—in which an event is observable if the environment can synchronously engage in it–are not directly relevant.

Unrolling the possible visible actions a process engages yields the *c-tree* (analogous to the "synchronization tree" for CCS) for the process. Two processes are considered *reactively equivalent* if, roughly, any visible action that one can do can be matched by the other in such a way that both processes end up in a state in which they remain reactively equal.

Note that the notion of visible action takes into account that telling (or asking) a constraint c is a silent action if the store is already strong enough to entail c: thus the agent $(X = 1) \rightarrow (X = 1 \wedge Y = 2)$ is reactively equivalent to the agent $(X = 1) \rightarrow Y = 2$. This highlights that the notion of communication in the cc languages is very different from that in CCS or CSP: in those systems each action is taken as atomic and does not interfere with any other actions the system may have taken in the past.

I once believed (in line with traditional dogma in concurrency theory) that reactive equivalence was the "finest" congruence that one could define on indeterminate cc agents. However, the emergence of linear logic has made it possible to treat indeterminacy on a logical footing, and it now seems plausible that there is an even finer congruence than bisimulation, namely linear logic equivalence. See Section 4 below for more details.

Graph Grammars and True concurrency semantics [MR91] develops a "true concurrency" semantics for CCP based on the notion of *graph grammars*. The state of the computation is described by a graph, with variables as nodes, and constraint tokens and agents as hyperarcs. *Both* the inference rules in the underlying constraint system, and (the user-specified) agents are expressed as *graph productions* which match subgraphs to other subgraphs, possibly introducing new nodes in the process, and deleting portions of the matched subgraph. A computation is then just a finite or infinite sequence of such graph rewriting steps. From such a computation, it is shown how to construct an *occurrence net*, a particularly simple kind of Petri-net which can express "causality" information. The move from occurrence-nets to partial orders is standard, though it depends on what is desired to be observed of the computation.

An attractive feature of this approach is that the "productions" implementing the underlying constraint system are treated on an equal footing with user-defined agents. Indeed, the constraint system itself can be seen as a closure operator on sets of tokens, ordered by the is-a-subset-of relation.

What remains unclear is the connection between the closure operator semantics (which abstracts from execution sequences), and the partial ordering semantics generated from an occurrence net. More work is needed to resolve this issue. Also unclear is the precise advantage gained by the more general true concurrency approach over the interleaving approach. Perhaps this will become clearer when one considers "refinable" basic actions (e.g., ask operations that themselves involve some computation, i.e., the so-called "deep guards", see [Sarng, Chapter 8]).

Based in part on this work, the authors and their colleagues have gone on to describe in [CMR92] an abstract machine, CHARM (for Concurrency and Hiding in an Abstract Rewriting Machine). Its distinguishing feature is the notion of partitioning the state of a process into a *global* (externally visible) and a local (hidden) part. Transitions preserve the global parts they match on; intuitively the global parts matched represent the structure of the underlying graph that is "asked for" (tested), but not consumed. CHARM is intended as a "unifying framework" for concurrency, since it is able to formalise in the same conceptual framework, the classical approach to graph grammars as well as the CCP paradigm; and its authors expect it will also be useful in formalizing the basic concepts in other concurrent paradigms such as CCS and CSP.

As we shall see below, the connection with graph grammars is particularly appropriate when one considers a visual syntax for (at least) the cc/Herbrand programming languages.

Future work. Among various aspects of concurrent constraint programming, semantic foundations seem to be the most well-developed. Perhaps the most glaring gap is that a comprehensive, systematic, theory of testing, with more powerful tests tied to finer congruences, has not yet been developed. Also, I believe that with the advent of linear logic, it may be possible to tie the notion of testing to the notion of recording as the denotation of a process a well-chosen *subset* of the logical formulas entailed by the process (viewed as a logical formula). If this works out, it will be a very beautiful way of exposing the logical structure of the programming language.

Also, I would hope that the work can serve as the foundation for the development of practical tools to help in partial evaluation, program transformation, and static analysis of programs.

3 Logical foundations of CCP

Where are the logical foundations of the (very operationally motivated) cc paradigm? [LS91] shows that a logical semantics may be given using a *formula-as-agent* and *proof-as-computation* interpretation of intuitionistic logic. The main underlying idea is to regard the steps of computation as expanding the frontier of a partial proof tree, whose leaves may be regarded as nonlogical axioms describing the residual computation, and whose root describes the initial state of the system. Each step of the computation unfolds a (leaf) nonlogical axiom into perhaps several other nonlogical axioms and some concrete proof structure. A completed computation corresponds to a completed proof.

For example, the basic determinate ask-and-tell `cc` languages can be described in intuitionistic logic thus. There are three syntactic categories of interest: Programs (P), Agents (A) and Goals (G). They are given by the syntax:

$$
\begin{array}{llll}
(Program) & P & ::= & H \rightarrow A & \text{– Clause.} \\
 & & & \mid \forall X\ P & \text{– Universal Closure.} \\
 & & & \mid P \wedge P & \text{– Multiple Definitions.} \\
 & & & & \\
(Agents) & A & ::= c & \text{– Tell.} & \\
 & & & c \rightarrow A & \text{– Ask.} \\
 & & & H & \text{– Procedure Call.} \\
 & & & A \wedge A & \text{– Parallel Composition.} \\
 & & & \exists X.A & \text{– Hiding.} \\
 & & & & \\
(Goals) & G & ::= & c & \text{– Primitive Test.} \\
 & & & \mid A \rightarrow G & \text{– Test an agent.} \\
 & & & \forall X.G & \text{– Generic Test.}
\end{array}
\tag{1}
$$

In the above syntax $\rightarrow$ is intended to be read as implication, and H ranges over the syntactic category of atomic formulas. Thus the fragment of logic specified above is in some sense "dual" to the usual definite clause logic underlying logic programming: a clause is of the form $H \rightarrow A$, *not* $H \leftarrow A$; one describes *necessary* rather than sufficient conditions. Intuitively, one thinks of specifying for each predicate some pieces of information that must follow (for some of the arguments to the predicate) when some other pieces of information are know to be true about some arguments to the predicate. Each predicate is thus thought of as a transducer of (partial) information. One thinks of the underlying engine as making explicit pieces of information that are implicit in the original presentation of an agent — as explicit as is needed to answer a (primitive) test.

For example, we can understand the usual directional program for append and naive reverse as corresponding to the following formulas

---

```
append(X, Y, Z) →
        ((X = [] → Y = Z)
         ∧ (cons(X) → ∃A∃B∃C.X = [A | B], Z = [A | C], append(B, Y, C))).
nrev(X, Y) →
        ((X = [] → X = Y)
         ∧ (cons(X) → ∃A∃B∃C.X = [A | B], nrev(B, C), append(C, [A], Y)).
```

Computation is initiated when the underlying execution engine is given a program and a goal. The operational semantics of the language is the usual one, with tells being accumulated in the store, and implication being treated as a blocking Ask: a configuration $(\Gamma, c \rightarrow A)$ can "be simplified to" (can make a transition to) the configuration (Γ, A) only if the constrains in Γ are strong enough to entail c. A minor modification is that execution terminates successfully if and when the store associated with the agent is strong enough to entail the constraint in the consequent of the goal.

The basic theorem connecting the operational semantics to logical notions is: a program presentation (P, G) succeeds *if and only if* the sequent $P \vdash G$ is derivable in intuitionistic logic (augmented with the axioms of the constraint system). Thus the usual operational imagery of store-as-constraint with Ask and Tell as basic operations can be understood merely as one way of implementing a complete theorem-prover for this fragment of logic.

More generally, these ideas lead to the formula-as-agent interpretation of a larger fragment of logic [LS91] — for example, one can "internalize" tests into the language by allowing agents of the form $G \rightarrow A$. One can allow for *recursive* testing (i.e., allow goals to be atoms, and allow programs to contain definite clauses specifying how such tests should be decomposed into more primitive tests). This shows how the notion of testing — a staple from the theory of concurrent systems — can be used to re-admit ("backward-chaining", definite-clause, constraint) logic programming into the CCP conceptual framework. In such a hybrid logical system, the "action" happening on the LHS of a sequent (in basic CCP) and on the RHS (in logic programming) can be combined smoothly. (From the perspective of AI-based reasoning system, what we have here is a smooth combination of constraint-based "backward-chaining" and "forward-chaining" systems.) With some care, it is possible to prove a much richer set of soundness and completeness theorems relating intuitionistic derivability to a simply motivated operational semantics.

Indeed, these ideas lead to a direct computational of a larger class of logics. In general, a subset of the formulas of the logic are viewed as computational agents. The left-rules in a sequent-style presentation of the logic are taken to specify the operational "heating" rules (in the style of [BB90]) which describe how complex agents may be decomposed into simpler agents, and the basic interactions betwen these agents. Thus, the operational derivability relation between configurations *qua* agents is related to the entailment relation between configurations *qua* logical formulas. In general, the logical counterpart of parallel composition is (seen to be) conjunction, of (ask-)prefixing is (a quantified version of) implication, and of hiding is existential quantification. Communication occurs (essentially) via constraint-imposition on first-order variables.

4 Extensions of the basic paradigm: Higher-order, Linear CCP

The logical interpretation of cc languages described above is proving to be very fruitful. In particular, it turns out that in order to handle indeterminacy logically, one needs to move on to the setting of linear logic. Once that is done, the path towards adding ω-order features to the language — allowing abstractions of processes to be passed as parameters — is also clear.

More generally, in [SL92] we show that a very large area of concurrency is opened up to the formulas-as-agent and proof-as-computation perspective by moving to the setting of a higher-order version of Girard's *linear logic* [Gir87]. Linear logic may be seen as arising from classical logic by dropping certain structural rules that allow formulas to be arbitrarily duplicated and eliminated during the course of a derivation. This causes the familiar conjunction and disjunction operations to split into two: the so-called *additive* versions that copy formulas, and the *multiplicative* versions which do not. This gives formulas the nature of *resources* that must be accounted for carefully during proof (computation). The lost power of the structural rules can be recovered locally by means of the "modalities" ! and ?. In particular, the formula $!A$ is allowed to be duplicated any number of times, derelicted to obtain A, and dropped altogether.[4]

Informal review of HLcc. The linear logic *formulas-as-resources* viewpoint may be reinterpreted as *formulas-as-communications*. Multiplicative conjunction ($\otimes$) is viewed as parallel composition. An (unbanged) atomic formula is regarded as an indivisible piece of information, a *message* placed in a pool of messages and agents. (Since $A \otimes A$ is not linearly equivalent to A, multiplicities of messages matter.) A banged atomic formula is regarded as a permanant (non-consumable), globally valid piece of information, that is, a constraint (multiplicities of constraints do not matter, logically $!c$ is linearly equivalent to $!c \otimes !c$). Linear implication ($\multimap$) is viewed as a *method* for transforming a communication into a collection of agents. Intuitionistic implication provides a more restricted form of method which can only be invoked with constraints. Additive Conjunction (&) provides external choice — the environment can cause the combination $(p \multimap A \,\&\, q \multimap B)$ to reduce to A or to B by supplying either p or q respectively. Universal quantification provides a parametrization mechanism: the method $\forall X.(p\ X) \multimap (a\ X)$ can reduce to $a\ (Y\ 3)$ in the presence of the message $p\ (Y\ 3)$. Existential quantification provides *hiding* — the ability to generate a new communication channel distinct from all others in the configuration. (Though it is outside the scope of this paper, a form of parallel search can be obtained through the additive disjunction $\oplus$.)

[4] See [Sce90,Lin92] for more motivation and tutorial introductions to linear logic.

Linear communication. The presence of unbanged atomic formulas ("linear tells") and linear implication ("linear asks") now allows for a direct representation of indeterminacy and state change: the configuration $p \otimes (p \multimap r) \otimes (p \multimap q)$ can evolve to (entail) either $a \equiv r \otimes (p \multimap q)$ or $b \equiv q \otimes (p \multimap r)$, but *not* to their tensor product $a \otimes b$. In Hewitt's terminology, this kind of indeterminism is called *arrival indeterminism*, since it depends on the order in which messages (here, p) are delivered to methods (here, $p \multimap r$, and $p \multimap q$).

Indeed linear asks and tells allow for a direct, powerful form of communication. An example should suffice to convey the flavor. A natural way to model Actor-style languages within the cc framework (see [SKL90,KS90a]) is to represent actor mailboxes as logical variables equated to *bags* of values. Sending a message, say 5, to an actor with associated channel X, then, corresponds to posting the constraint $5 \in X$. The actor suspends, asking whether there is a message on X; when activated, it pops a message off the bag and recurs with the tail of the bag.

However, such a translation does not work: two different actors may wish to communicate the *same* message to an actor — hence they would both post, say, the constraint $5 \in X$. Operationally we would like *both* messages to be delivered to (the actor) X; in (intuitionistic/classical) logic, however, this situation cannot be distinguished from that in which *only one* message was sent, since in these logics $a \wedge a$ is logically equivalent to a.

The problem does not arise in linear logic since multiplicities matter, and the above translation works correctly. Since a configuration is *itself* a multiset of agents, we need not use constraints such as $5 \in X$. Rather, just atomic formulas of the form $X : 5$ suffice — here ":" is a binary predicate that has no special properties (built-in inference mechanism). An atom $(b \equiv) X : 5$ is, by convention, read as "5 is a message on channel X". An agent can read a message on the channel, and forward it on another channel Y by means of the method: $(a \equiv) \forall M. X : M \multimap Y : M$. As we shall see, the agent $a \otimes b$ can evolve into the agent $Y : 5$, as expected. Note that simultaneously *many* agents can send to the same channel, and many can read from the same channel (as with $p \multimap q$ and $p \multimap r$ above): the underlying notion of communication is *many-to-many* as in Linda-like languages, rather than many-to-one, as in Actor languages. Summarizing: Agents have access to a communication channel. On this channel, an agent can either post a *message* or a *method*. A method has the ability to suspend until a message matching a specified pattern arrives on the channel. Once activated, a method may, recursively, create new channels, and post messages and methods on new or old channels.

Note that *state* and *state change* is just the flip side of this ability to receive messages and consume them. The following example will demonstrate a number of programming techniques simultaneously. One may set up a `counter_creator` agent as follows:

$$
\begin{aligned}
\mathtt{A} \equiv \forall \mathtt{counter}.(&\mathtt{counter\_creator} : \mathtt{counter} \multimap \\
&\exists \mathtt{value}(\mathtt{value} : 0 \otimes \\
&\quad \forall \mathtt{m}(\mathtt{counter} : \mathtt{m} \multimap \\
&\qquad (\forall \mathtt{val}\ (\mathtt{m} : (\mathtt{inc\ val}) \multimap \forall \mathtt{v}\ (\mathtt{value} : \mathtt{v} \multimap \mathtt{value} : (\mathtt{v} + \mathtt{val}))) \\
&\qquad \& \forall \mathtt{val}\ (\mathtt{m} : (\mathtt{value\ val}) \multimap \forall \mathtt{v}\ (\mathtt{value} : \mathtt{v} \multimap \mathtt{value} : \mathtt{v} \otimes \mathtt{val} : \mathtt{v})) \\
&\qquad \& \forall \mathtt{val}\ (\mathtt{m} : (\mathtt{dec\ val}) \multimap \forall \mathtt{v}\ (\mathtt{value} : \mathtt{v} \multimap \mathtt{value} : (\mathtt{v} - \mathtt{val})))))).
\end{aligned}
$$

Informally, such an agent responds to a message `counter` sent on the `counter_creator` channel as follows: it sets up a new local communication channel, `value`, initializes it with the message 0, and installs a method on the *input* message channel, which suspends until a message `m` arises, and then suspends until `m` (which is itself a channel), contains a parametrized `inc` or `dec` or `val` message. In the `inc` case, it reads the value `M` on its private channel (thereby deleting the value), and sends the message $\mathtt{M} + \mathtt{val}$ on `value`. Similarly, in the other cases.[5] Note: an assignable memory location is merely a channel with a single outstanding message; incoming channels are just like any other channels ...it is possible to both read and write (post messages and methods on) them; the effect of lexically scoped, shared, encapsulated, concurrently updatable state variables is created through the alternation of universal and existential quantifiers; the usual logic programming ideas of "incomplete" messages are still available; and in the spirit of Hewitt's slogan, control-structures emerge from patterns of communications.

The higher-order system. For all its power, however, it is not possible in the first-order Lcc to pass *processes* as parameters. At best one can pass names of (channels to) processes. One cannot obtain unknown processes from communication channels, destructure them, combine them, run them, apply them to other unknown processes or otherwise treat them as first-class citizens. This motivates our final move, then, to higher-order linear logic.

The higher-order version of linear logic we consider is a straightforward adaption of a presentation of higher-order intuitionistic logic using Church's simple theory of types (the details are contained in [SL92]). Intuitively, one may understand such a logic as obtained from first-order linear logic by replacing first-order terms with simply typed λ-terms. One assumes that the set of types comes equipped with a type o for propositions, and a type ι for individuals; the linear connectives can then all be introduced as logical constants (functions) over o. (Similarly, for each type α, the existential quantifier $\Sigma^{\alpha}_{(\alpha\rightarrow o)\rightarrow o}$ and the universal quantifier $\Pi^{\alpha}_{(\alpha\rightarrow o)\rightarrow o}$ are also provided. This allows, then, the expression of terms such as

$$(\mathrm{t} \equiv)\lambda \mathrm{a}_\iota \lambda \mathrm{b}_\iota \lambda \mathrm{c}_\iota . \Pi \lambda \mathrm{y}_o (\mathrm{a} : \mathrm{y} \multimap \Pi \lambda \mathrm{z}_o (\mathrm{b} : \mathrm{z} \multimap \mathrm{c} : (\mathrm{y} \otimes \mathrm{z})))$$

---

[5] As described here, all these actions will happen *exactly once*. [SL92] describes how recursion and repetition are programmable in the higher-order language.

which can be viewed as an abstraction which when applied to three communication channels a, b and c yields a process that will accept an arbitrary process y on a and z on b, and send their parallel composition $y \otimes z$ on to c. Of course t is itself just a term and can be passed around in messages from agent to agent; in that case the receiving agent would need to use a universal quantifier at a functional (higher-order) type to receive the message.

In general, higher-order quantification is an exceedingly powerful mechanism, and developing practical programming languages with such features is a delicate task. The main restriction placed on these quantifiers in HLcc involves limitations on the use of Π, restricting occurrences of Π to immediately precede $\multimap$, with further restrictions which effectively ensure that it is not possible to universally quantify on the first argument (the "channel") in a message $x : t$, and no universally quantified unguarded agents will ever appear during execution. (An agent of the form $\Pi : \lambda x.x$ is an agent which can perform *any* task, provide *any* information to other agents, etc.; such an agent is computationally disastrous.)

For pragmatic reasons, it may be desirable to impose syntactic restrictions which disallow existential quantification at functional types and/or to disallow in the head logical constants or λ-abstractions or multiple occurrences of variables (see, e.g., [Wad91]). This can obviate the need for higher-order unification or matching at run-time; however, the higher-order structure left in the programming language is still adequate to construct, assemble, communicate and use higher-order expressions.

5 Concrete cc languages

We turn finally to some work in the last few years on the design and implementation of concrete programming languages in this framework.

5.1 Distributed cc languages: Janus.

In the context of distributed computation, a fundamental problem with cc languages (the *multiple teller* problem) is that a computation may have multiple producers of partial information for the same set of variables. Unless these producers are coordinated in some way with respect to each other, they may produce conflicting constraints on the same variable, thus causing the system to crash. For example, one agent may constrain a variable `X` to be equated to `3` and another may constrain it to be `4`.

Languages such as Herbrand used the notion of atomic publication to circumvent the problem: An agent must augment the constraints in the store atomically, failing if the store is incompatible with the new information. However, this notion is far too heavy-weight to serve as a foundational communication mechanism in a distributed setting Other cc/Herbrand languages such as Parlog, FGHC and Strand deal with the multiple teller problem by ignoring it! They trust that the programmer will organize computations in such a way that the computations will never generate conflicting constraints at run-time. If this trust is violated then the entire computation will abort with a logical error.

These motivations led to the design of Janus [SKL90]. The fundamental intuition is that mechanisms should be devised such that computation is not forced to abort at run-time because the store is rendered inconsistent. Second, the communication mechanism should naturally reflect the notion of asynchronous message passing, to be compatible with basic distributed computation concerns. That is, it should be possible for agents at a site to progress by publishing information without having to coordinate with agents at other sites. (This desideratum is even more important if it is desired that computations in other languages, built using different abstractions, should be able to *inter-operate* smoothly with computations in these languages.)

Finally, if the languages are to be widely used as procedural languages there should at least be the potential for a performance-efficient implementation competitive with that of more conventional distributed assignment-based languages. Even more strongly, we believe that uni-processor implementations of such languages should be comparable in performance with the implementations of sequential, imperative languages.

Janus attacks this problem on two fronts. First, we identified a constraint system, suitably rich for procedural programming, which possesses certain "independence" properties and naturally supports many-to-one communication. Suppose `X` and `Y` are two variables unconstrained in the current store, and an agent decides to post the (consistent) constraint `X=s`, for some term `s` possibly containing `Y`. Then, intuitively, for the constraint system to be implementable well in a distributed setting it should not be possible to construe this imposition as a constraint on `Y`. That is, more generally, if c_1 is a constraint of the form $\wedge_{i=1}^{n} X_i = s_i$ and c_2 of the form $\wedge_{j=1}^{m} Y_j = t_j$, where the X_i and Y_j are distinct, then $c_1 \wedge c_2$ should be consistent iff c_1 and c_2 are consistent separately. The Herbrand constraint system does not satisfy this property: if the constraint `X=f(Y)` is imposed, then this restricts `Y`; for example, the constraint `Y=X` cannot be imposed. Part of our task was to design a constraint system, $\mathcal{J}$, closely related to Herbrand, which satisfies such properties. Essentially, the constraint system is defined over infinite trees [Col84]; but matters are complicated due to the necessity of introducing "hyper-multisets" in order to cater for many-to-one communication. In designing $\mathcal{J}$ our paramount concern had been that it should be efficiently implementable in a distributed setting, expressive enough to allow straight-forward and elegant formulation of standard algorithms over standard data-structures such as arrays and lists, and should be such that with an understanding of some basic representation decisions made by an implementation, a programmer can fairly accurately predict the cost of execution of various programs.

Second, we have exploited the idea of *syntactic control of interference*. Compile-time checkable restrictions are introduced which ensure that if the current store is consistent and a goal reduces via a program clause, then the resultant store will be guaranteed to be consistent, *without having to check for consistency.* This is achieved by exploiting a remarkably simple idea that I first stumbled across in 1987 when studying the notion of broadcast-free communication.

Say a clause is binary if any variable occurs free at most twice in the clause. In such a clause, a variable may be considered as a "point-to-point" communication channel because messages generated on it from one occurrence only need to travel to one other occurrence. Now, observe that the resolvent of two binary clauses is also binary! ("Two" is a magic number here — from three one may generate as many occurrences as one wants.) This means that if the source program clauses satisfy this condition, then any configuration generated at run-time will also satisfy this condition. All that remains then is to (a) devise a scheme which ensures that exactly one of these occurrences (the *teller*) can be used to impose a constraint on the variable — the other occurrence (the *asker*) is then used to "receive" information (b) ensure that in this scheme "tell-rights" can be communicated on channels, and (c) determine if this scheme is powerful enough to express the characteristic idioms of concurrent logic programming. All three goals were achieved in [SKL90], adapting ideas from [Hir86].

A major opportunity offered by Janus is that of compile-time garabage-collection. Because any variable has at most two occurrences, once one occurrence of a variable receives a term, it "knows" that there are no other references outstanding to that term (the other occurrence of the variable has been "used up" in binding the variable to the term). Therefore, a compiler can generate code knowing that whenever a data-structure in an atom has been successfully matched, the data-structure is garbage and can be recycled.

The hypothesis that it will be possible to implement Janus very efficiently on conventional uniprocessor machines has to some extent been borne out by the Janus-to-C compiler described in [GDD92]. As in the implementation discussed in this thesis, a basic idea is to try to use a stack as far as possible so as to execute code that does not need to suspend at least as efficiently as Prolog would. An attempt is made at compile-time to figure out an execution order for goals that schedules producers before consumers, so that the number of suspensions is minimized. A call-frame is copied out onto the heap in a suspension record only when executing the code corresponding to it checks for a binding on an asker that is not there yet. The simple cases of direct suspension (a process suspending on a single variable) are optimized. Through a combination of these ideas, and very careful engineering, it has been possible to obtain an implementation which is considerably faster than the implementation of any other logic programming language. ([GDD92] contains comparisons.)

A "Quick-and-Dirty" implementation of Janus is also available on top of Prolog [Deb92]. The implementation essentially uses the scheme described in this thesis to manage synchronization. It is able to use the freeze operation of the underlying Prolog engine since Janus does not have deep guards — hence the guard isolation problem (see [Sarng, Chapter 9]) does not arise. Rather creditable performance numbers are reported, considering that Janus is being translated into a high-level programming language.

Another implementation, based on a byte-code emulated abstract machine is, is being developed by Jacob Levy.

5.2 Linear Janus.

Linear Janus [Tse92] is a first-order language on one extreme end of the spectrum of Lcc languages. It is the instance of Lcc obtained by having no built-in constraint system whatsoever, relying only on linear ask and tell communication. Indeed, no atomic formulas other than ":" are allowed either — thus "all one can do" in such a language is post messages on channels or post methods on channels.[6]

Computationally, Linear Janus is a cleaned up and considerably simplified version of Lucy, the "missing link" between concurrent logic programming and actor languages described in [KS90a]. As such it is in a long tradition of languages arising from the connection between concurrent logic programming languages and Actor languages. In particular, the design decision not to build in any constraint system was made precisely to test out a significant hypothesis underlying actor-based systems: it is possible to conceptualize of computations purely in terms of actors at arm's-length relationship with each other. There are to be no built-ins or primitive operations with a special status.

π-calculus. An important connection is with the π-calculus. Linear Janus is very close to the "asynchronous fragment" of the π-calculus [MPW89]. Table 1 demonstrates, how a subset of the π-calculus may be embedded into (first-order) Lcc. Channels are treated as first-order variables, and the rest of the translation follows naturally. The seemingly mysterious notions of "extrusion" and "intrusion" of scope are seen as well-known operational manifestations of the logical properties of the first-order quantifiers.

Indeed, one of the strongest arguments for our approach is that the design of the language, together with its rules for execution, are immediately suggested by the underlying logic ...one does not have to motivate them just through completely operational arguments, which can sometimes be unconvincing because there may appear to be more than one way of capturing some operational phenonmenon. In particular, there are already several variants of the π-calculus [MPW89,Mil90a,Mil91], each emphasizing various operational aspects; it may not be clear which variant is to be preferred for which purpose. It is this *architectural* view of logic that I think will, in the long run, be one of the more valuable contributions of this line of research to the theory of concurrent systems.

[6] In first-order Lcc, the higher-order structure can no longer be used to obtain the effect of recursion. Instead, it is much simpler to augment the language with a "replication" operator (!) following [Mil90a]: the formula $!A$ is effectively equivalent to the formula $A \otimes (!A)$.

$$
\begin{array}{rcl}
[0] & = & 1 \\
[(y)P] & = & \exists y.[P] \\
[\bar{x}y.0] & = & x : y \\
[x(y).P] & = & \forall y.x : y \multimap [P] \\
[P \mid Q] & = & [P] \otimes [Q] \\
[A(X_1, \ldots, X_n)] & = & (: A\ X_1 \ldots X_n) \\
[A(X_1, \ldots, X_n) = P] & = & !\,(\forall X_1\ \ldots \forall X_n(: A\ X_1 \ldots X_n) \multimap [P]))
\end{array}
$$

Choice non-determinism ("+") is not handled; a guarded version can be translated into "methods".
Note that the structural rules of [Mil90a] are immediately verified. That is, $P \equiv Q$ implies $\vdash [P] \circ\!\!-\!\!\circ [Q]$.

Table 1: Translation of the input-guarded π-calculus to Lcc

It is possible to translate a larger fraction of the entire language (e.g., with full tell-prefixing) — at the cost of implementing a synchronous π-calculus transmission with a few-step transmit/acknowledge protocol.

Lambda calculus. Perhaps the most remarkable connection is with the lambda calculus. The translation shown below effectively shows how to embed the λ-calculus in a very simple fragment of first-order linear logic. John Lamping has pointed out that the translation also tells us how to embed the λ-calculus into the corresponding fragment of intuitionistic logic.

We focus here on the lazy version of the lambda-calculus [AO89]. The translation can be seen as the Lcc analogue of the translation in [Mil90a]. Intuitively, an application is treated as a parallel composition, and the operand and the argument are treated as separate processes communicating on a channel hidden from the outside world. A λ-abstraction is seen as a server waiting for a message to come down its specific communication channel (such messages are generated by application). β-reduction is reflected by the universal instantiation and (linear) *modus ponens* underlying the operational semantics of the Lcc languages. For M a lambda-term, and z a Linear Janus variable, we define $\langle M \rangle_z$ by:[7]

$$
\begin{array}{l}
\langle x \rangle_z = (: x\ z) \\
\langle \lambda x\ M \rangle_z = \forall x \forall y\ (: z\ x\ y) \multimap \langle M \rangle_y \\
\langle MN \rangle_z = \exists x \exists y (\langle M \rangle_x \otimes (: x\ y\ z) \otimes (!\forall s\ (: y\ s) \multimap \langle N \rangle_s))
\end{array}
\qquad (2)
$$

The translation can be read thus: the value of M is z iff $\langle M \rangle_z$, where $\langle M \rangle_z$ is read as a linear formula, with $(:\ x\ y\ z)$ being read as "the value of x applied to y is z", and $(:\ x\ z)$ as "the value of x is z". For instance, the clause for $\langle MN \rangle_z$ is read as: "the value of $\langle MN \rangle$ is z iff there exists an x and y such that

[7]Note that we use two constants for message transmission: $:_{\iota \to \iota \to o}$ and $:_{\iota \to \iota \to \iota \to o}$.

the value of M is x and the value of x applied to y is z, and for any s, the value of y is s implies that the value of N is s".

Concretely, the agent $\langle \lambda x\ x \rangle_z$ is just:

$$\forall x \forall b\ (: z\ x\ b) \multimap (: x\ b)$$

That is, $\lambda x\ x$ is viewed as a server that waits for an argument x, and then asserts that the result b is just x. Logically, one reads it as saying that the value of $\lambda x\ x$ is z iff for all x and b, the value of applying z to x is b implies that the value of x is b.

Implementation. An implementation of Linear Janus is currently under way, based on translation to an abstract machine which is implemented with an emulator written in C. Initial performance results are very promising, but indicate that the compiler has to be sensitive to various patterns of communications on channels. Indeed, channels are the *only* communication mechanism in Linear Janus and hence serve the role of "pointers" as well as many many-to-many communication links.

5.3 Languages for combinatorial exploration

Design of cc(FD). We are in the process of designing and implementing a "rational reconstruction" and extension of the constraint logic programming language CHIP [DvH+88]. The following description of the goals of the project and the strategy used is adapted from [HSD92].

The feasibility and practicality of a declarative language supporting both consistency techniques and nondeterminism was demonstrated in the CHIP system [Hen89,DvH+88]. Solutions for a large variety of combinatorial problems — test-generation [Sim88], car-sequencing [DSH88], microcode labelling [DSH90], scheduling and cutting stock [Hen89] — were developed quickly, and were comparable in efficiency (within a constant factor) to specialized algorithms using the same approach. CHIP supported consistency techniques mainly by providing a number of primitive constraints, such as equations, inequalities, and disequations involving variables ranging over finite subsets of natural numbers.

Though successful, CHIP has a number of limitations, the most important being its lack of extensibility. The programmer has to recast non-primitive constraints in terms of existing constraints, is unable to define new primitive constraints and is unable to cope adequately with disjunctions of constraints or the definition of incomplete constraint solvers. As a result, non-primitive constraints are often recast in terms of more basic variables (e.g. Boolean variables) and/or are used to make choices. Both alternatives decrease pruning, increase the search space, and entail a substantial loss in efficiency compared to procedural languages. The problem is not specific to CHIP but present in all CLP languages which are all based on a black-box constraint solver that the programmer cannot extend or modify.

We propose to remove the above limitation by taking a different approach. Our thesis is that consistency techniques can be supported in a much more fundamental way by providing general-purpose, efficiently implementable, declarative operations in the programming language that capture essential algorithmic techniques embedded in the implementation of such systems. To support this thesis, we introduce four new constraint language ideas: (1) *indexical constraints*, related to a restricted use of universal quantification, (2) *cardinality operators* with indexical bounds, related to the threshold operator of propositional logic [HD91], (3) *determinate disjunction* (see Section 2.1) related to intuitionistic disjunction, and (4) *extended Asks*, related to intuitionistic implication. We instantiate these ideas in the programming language cc(FD), incorporating a very simple constraint system over finite systems, FD. Through these combinators, we show that cc(FD) supports the basic reasoning principles implemented in systems such as ALICE [Lau78], REF-ARF [Fik68], CHIP in a very simple and flexible way.

The main advantages of such an approach are generality, extensibility, efficiency and semantic simplicity. The constructs introduced are *general-purpose*: they are definable on very general notions of constraint systems [SRP91], including rational arithmetic, Booleans and first-order terms. They are powerful enough that the underlying constraint-solver can be extremely simple (e.g., it may provide only unary constraints). The additional functionality provided by more powerful constraint-solvers can be obtained by means of user-level programs. This significantly reduces the black-box effect. The programmer can define his own constraints (to the extent of the reasoning principles supported) without sacrificing the declarative reading of the programs. Since the embedded constraint-solver is extremely simple, it can be very efficiently implemented. The combinators themselves are sufficiently "low-level" as not to introduce overhead (in time or space) compared to procedural implementation of the reasoning principles supported. And finally, the abstract semantics of these operators can be presented in a very simple way, using the notion of closure operators.

Implementation. cc(FD) is being implemented from scratch in C and C++, using a variant of the Warren Abstract Machine as the target language for the compiler. The implementation is expected to be available for general use in a few months.

Andorra Kernel Language Work on the Andorra Kernel Language [HJ90, JH91] derives its motivation from two sources. The first source is CCP and the second is the "Andorra" execution strategy due to David H.D. Warren [War87]. In addition, it is animated by the explicit desire to be "upwards" compatible with Prolog; being based on CCP ideas (with eventual publication) it is obviously upwards compatible with GHC.

This motivation has led to a special emphasis on the design and development of implementation mechanisms for "deep guard" computations, since, in general, Prolog allows arbitrary computation to the left of a "cut" operation in the body of a clause. The constructs are motivated primarily from implementation concerns. This means that there is some gap between the operational semantics (which is expressed via a set of rewrite rules on configurations represented as trees, in a manner reminescent of [Sar87]) and the logical interpretation of the program. I believe that this gap can be bridged using the notion of "recursive testing". As above, to understand "committed choice" logically, one has, of course, to move to linear logic.

[JH91] shows how various programming techniques available within both the committed choice framework and Prolog can be expressed within AKL. In addition, several new techniques for combinatorial control are described.

Implementation. Currently an experimental version, AKL/0.0 is available from SICS. This version supports Ask and Tell operations on the Herbrand constraint system and feature structures. The system includes a garbage collector, an interactive top-level written in AKL, a rudimentary compiler written in Prolog (produces code for an abstract machine emulated in C), an X-based graphic interface, support for foreign subroutine calls, and a notion of reflective AKL-engines to encapsulate nondeterminism, failure and deadlock. The purpose of the prototype was to get a quick implementation for studying the functionality of the system; it is not optimized for efficiency. Currently, it is about 3-4 times slower than SICStus Prolog on quick-sort and naive reverse. (On the other hand, it has determinacy propagation built-in, unlike Prolog, and this usually gives a performance improvement in search problems.)

Philippe Codognet has modified a version of AKL/0.0 to incorporate some ideas related to intelligent backtracking.

Currently, SICS is implementing the next, version AKL/1, designed to optimize the implementation of "flat" guards. The system compiles to abstract machine instructions that are suitable for further compilation to native code, and for direct emulation. There are plans to produce from this system a native code compiler for the MIPS processor, and a parallel implementation for Sequent and SUN multiprocessors.

5.4 Completely Visual Programming: the Pictorial Janus project

In the summer of 1988 and early 1989, Ken Kahn and I — with the crucial help of Randy Smith — designed a completely visual syntax for Herbrand [KS90b]. (In retrospect, some aspects of the syntax bears a remarkably close relationship to the graph grammar representation of Herbrand programs.) Each atomic formula (atom) is represented as a closed contour with *terminals* on its surface corresponding to the arguments of the atom. Terms are closed contours with a distinguished terminal *inside* the contour. These terminals are connected to each other by means of *wires*, representing equality links. Similarly syntax is specified for ask and tell agents, for recursion and for non-deterministic selection. The syntax is remarkably economical and in the case of some complex programs with a large number of variables, considerably less cluttered and easier to understand.[8]

An important aspect of the visual notation is that *exactly* the same visual terms are used to represent a program, a state of the execution, and the evolution of the program during execution. Also, *no* aspect of the computational machinery used to implement the language is hidden: every aspect of the system which is relevant to describing its behavior (evolution over time) is explicitly represented in the visual syntax. In very real terms, the *picture* is the program.

Another important aspect of the design of the visual syntax is that very few "visual elements" are used up in specifying the notions of ask- and tell-agents and rules. In particular, the only important aspect of a picture is its constituents and their interconnections: the shape, size, color, orientation etc. of the constituents is completely irrelevant. This leaves these visual elements free for the use of the programmer in whichever way he or she deems appropriate for the application at hand.

Since 1989, these ideas have become far more concrete, and a prototype implementation of some aspects of a completely visual programming environment are in place [Kah92a,Kah92b] in a system called Pictorial Janus. Programs can be drawn using any visual editor capable of producing PostScript code. This code is then run with a modified PostScript printer to produce a representation of the picture in graphical terms. A visual parser accepts such an input, recognizes in it terminals, contours, agents and constraints, and produces an internal representation that contains graphical information about the layout of the program (e.g., color, shape, line weight etc.) as well as the actual Janus program sketched out by the user. This internal representation can then be fed into a Janus implementation. A movie can then be made describing the various stages in the evolution of the initial configuration, with aspects such as rule-matching completely animated. In fact, mechanisms also exist to associate each aspect of the animation of the program (e.g., rule expansion) with sound, so that one can actually *hear* the sound of the program executing during playback.

Pictorial Janus has been used to produce a wide variety of animations which include concurrent algorithms, cc programming techniques, the Object Management Group Common Object Request Broker Architecture, the Pictorial Janus system architecture, a parallel distributed text retrieval system, a work-flow example (a publication clearance process), the value of cooperation among computational agents, and real-time controllers. Substantial work has been done in applying this technology to the qualitative models of a recirculating document

[8] Indeed for several months in 1988 I had contemplated redoing all the programs in my thesis in visual notation. I finally deemed it too radical for a document that I was most interested in finishing very quickly!

handler.

The main attraction of the completely visual formalization is that it makes the Herbrand/Janus computation model accessible to a much wider audience, since it succeeds in describing the basic elements of the compuation model in purely pictorial terms (via the notion of parts of a picture matching other parts of a picture). There is considerable hope that, in the tradition of Logo, a somewhat simplified programming language in the same style may be useful for teaching the basic ideas of concurency and programming languages to young children. Similarly, we believe that completely visual syntax can be devised for other cc languages with different built-in constraint systems.

6 Conclusion

This is an exciting time. The development of HLcc seems to indicate that we are close to a framework where some of the most important "declarative" paradigms — functional programming and constraint programming, including its progenitor, logic programming — can be combined smoothly with work on process algebras. Such a framework would be a powerful host for languages for symbolic computation that are at once concurrent, asynchronous, higher-order, and based on a very sophisticated form of communication. It would enable the creation of semantically-based tools to help a programmer design, construct, analyse, type-check, verify and compile programs.

But a gleam in the eye is very different from a theory in hand, or even an implementation in the bush. Considerable technical work lies ahead — at the end is a beginning.

References

[AO89] S. Abramsky and C.-H.L. Ong. Full abstraction in the lazy lambda-calculus. Technical report, Dept. of Computing, Imperial College, 1989. To appear in Information and Computation.

[BB90] Gerard Boudol and G. Berry. The Chemical Abstract Machine. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 81–94. ACM, 1990.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the Association for Computing Machinery*, 31(3):560–599, July 1984.

[CMR92] Andrea Corradini, Ugo Montanari, and Francesca Rossi. CHARM: Concurrency and hiding in an abstract rewriting machine. In *Proceedings of the Fifth Generation Computer Systems Conference*, 1992.

[Col84] A. Colmerauer. Equations and inequations on finite and infinite trees. In *2d International Conference on Fifth Generation Computer Systems*, pages 85–99, 1984.

[dBKPR90] Frank S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures in a paradigm for asynchronous communication. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of CONCUR '90*, volume 458 of *Lecture Notes in Computer Science*, pages 99–114. Springer Verlag, 1990.

[dBP90] Frank S. de Boer and C. Palamidessi. Concurrent logic languages: Asynchronism and language comparison. In *Proceedings of the North American Conference on Logic Programming*. M.I.T. Press, 1990.

[dBP91a] F. S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In *Proceedings of TAPSOFT/CAAP*, volume 493 of *Springer-Verlag Lecture Notes in Computer Science*, pages 296–319, 1991.

[dBP91b] Frank S. de Boer and C. Palamidessi. On the asychronous nature of communication in concurrent logic languages: a fully abstract model based on sequences. In *Proceedings of CONCUR '90*, 1991.

[dBP92] F. S. de Boer and C. Palamidessi. A process algebra of concurrent constraint programming. In *Proceedings of the Joint International Logic Programming Conference/Symposium*, 1992. To appear.

[Deb92] Saumya K. Debray. QD-Janus: A sequential implementation of Janus in Prolog. Technical report, Department of Computer Science, University of Arizona, 1992. Submitted for publication.

[DSH88] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car Sequencing Problem in Constraint Logic Programming. In *European Conference on Artificial Intelligence (ECAI-88)*, August 1988.

[DSH90] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):75–93, 1990.

[DvH+88] M. Dincbas, P. van Hentenryck, H. Simonis F. Berthier. The constraint logic programming language chip. In *Proceedings of the FGCS Conference*, November 1988.

[Fik68] Richard Fikes. *A Heuristic Program for Solving Problems Stated as Nondeterministic Procedures*. PhD thesis, Carnegie Mellon University, 1968.

[GCLS88] Rob Gerth, Mike Codish, Yossi Lichtenstein, and Ehud Shapiro. A fully abstract denotational semantics for Flat Concurrent Prolog. In *LICS 88*, 1988.

[GDD92] D. Gudeman, K. DeBosschere, and S.K. Debray. jc: An efficient and portable sequential implementation of Janus. In *Proceedings of the Joint International Logic Programming Conference/Symposium*, 1992. To appear.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[GL90] Maurizio Gabbrielli and G. Levi. Unfolding and fixpoint semantics of concurrent constraint logic programs. Technical report, University of Pisa, 1990.

[GMS89] Haim Gaifman, Michael J. Maher, and Ehud Shapiro. Reactive behavior semantics for concurrent constraint logic programs. In *North American Logic Programming Conference*. MIT Press, October 1989.

[HD91] P. Van Hentenryck and Y. Deville. The Cardinality Operator: A new Logical Connective and its Application to Constraint Logic Programming. In *Conference Proceedings of the Eighth International Confernce on Logic Programming*, June 1991.

[Hen89] Pascal Van Hentenryck. *Constraint satisfaction in logic programming.* Logic Programming Series. MIT Press, 1989.

[Hir86] Masahiro Hirata. Programming language doc and its self-description, or, x = x considered harmful. In *3d Conference Proceedings of Japan Society for Software Science and Technology*, pages 69–72, 1986.

[HJ90] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its computation model. In *Proceedings of the Seventh International Conference on Logic Programming*, June 1990.

[Hoa89] C.A.R. Hoare. A theory of conjunction and concurrency. Oxford PRG, May 1989.

[Hoa90] C.A.R. Hoare. Let's make models. In *Proceedings of CONCUR 90*, August 1990.

[HSD92] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. cc(FD): Constraint programming over finite domains. Technical report, Computer Science Department, Brown University, forthcoming 1992.

[JH91] Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium*. MIT Press, 1991.

[Jos90] Mark B. Josephs. Receptive process theory. Technical report, Programming Research Group, Oxford University, July 1990.

[JPP89] R. Jagadeesan, P. Panangaden, and K. Pingali. A fully abstract semantics for a functional language with logic variables. In *Proceedings of LICS*, page ??, June 1989.

[JSS91] R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic non-determinism in concurrent constraint programming. Technical report, System Sciences Laboratory, Xerox PARC, January 1991.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Proceeedings of IFIP Congress 74*, pages 471–475., August 1974.

[Kah92a] Kenneth M. Kahn. Concurrent constraint programs to parse and animate pictures of concurrent constraint programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*. ICOT, June 1992.

[Kah92b] Kenneth M. Kahn. Towards visual concurrent constraint programming. Technical report, SSL, Xerox PARC, 1992. Contains a play written and directed by Ken Kahn, performed during San Francisco Bay Area ACM SIGCHI Seminar.

[KS90a] Kenneth M. Kahn and Vijay A. Saraswat. Actors as a special case of concurrent constraint programming. In *Proceedings of the Joint Conference on Object-Oriented Programming: Systems, Languages, and Applications and the European Conference on Object-Oriented Programming*. ACM Press, October 1990.

[KS90b] Kenneth M. Kahn and Vijay A. Saraswat. Complete visualization of concurrent programs and their execution. In *Proceedings of the IEEE Workshop on Visual Programming*. IEEE, October 1990. Also SSL technical report, Xerox PARC, August 1989.

[Lau78] J.-L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artifical Intelligence Journal*, 10:29–127, 1978.

[Lev88] Giorgio Levi. Models, unfolding rules and fixpoint semantics. In *Proceeedings of the Fifth International Conference and Symposium on Logic Programming, Seattle*, pages 1649–1665, August 1988.

[Lin92] P. Lincoln. Linear logic. *SIGACT Notices*, 23(2):29–37, Spring 1992.

[LS91] Patrick Lincoln and Vijay A. Saraswat. Proofs as Concurrent Processes: A Logical Interpretationfor Concurrent Constraint Programming. Technical report, Systems Sciences Laboratory, Xerox PARC, November 1991.

[Mil80] R. Milner. *A calculus of communicating systems*. Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[Mil90a] R. Milner. Functions as processes. In M. S. Paterson, editor, *The Seventeenth International Colloquium On Automata Languages And Programming*, pages 167–180. Springer-Verlag, 1990. Lecture Notes In Computer Science 443.

[Mil90b] R. Milner. *Handbook of Theoretical Computer Science, Vol. B*, chapter Operational and Algebraic Semantics of Concurrent Processes. Elsevier Science Publishers B.V., 1990.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. LFCS Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[MPW89] R. Milner, J. G. Parrow, and D. J. Walker. A calculus for mobile processes, part i and ii. LFCS Report ECS-LFCS-89-85, University of Edinburgh, 1989.

[MR91] Ugo Montanari and Francesca Rossi. True concurrency semantics for concurrent constraint programminmg. In V. Saraswat and K. Ueda, editors, *Proc. of the 1991 International Logic Programming Symposium*, 1991.

[OH86] E.-R. Olderog and C.A.R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica*, 23:9–66, 1986.

[PSSS91] P. Panangaden, Vijay A. Saraswat, Phillip J. Scott, and Robert Seely. A Hyperdoctrinal View of Constraint Ssystems. To appear in Proceedings of 1992 Rex Workshop, 1991.

[Sar85] Vijay A. Saraswat. Partial correctness semantics for $\mathbf{cp}(\downarrow, |, \&)$. In *Proceedings of the FSTTCS Conference*, number 206 in Lecture Notes in Computer Science, pages 347–368. Springer-Verlag, December 1985.

[Sar87] Vijay A. Saraswat. The concurrent logic programming language **cp**: definition and operational semantics. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 49–62. ACM, January 1987.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989. To appear, Doctoral Dissertation Award and Logic Programming Series, MIT Press, 1992.

[Sar92] Vijay A. Saraswat. The Category of Constraint Systems is Cartesian-closed. In *Proc. 7th IEEE Symp. on Logic in Computer Science, Santa Cruz*, 1992.

[Sarng] Vijay A. Saraswat. *Concurrent constraint programming languages.* Doctoral Dissertation Award and Logic Programming Series. MIT Press, 1990, forthcoming.

[Sce90] A. Scedrov. A brief guide to linear logic. *Bulletin of the European Assoc. for Theoretical Computer Science*, 41:154–165, June 1990.

[Sco82] Dana S. Scott. Domains for denotational semantics. In *Proceedings of ICALP*, 1982.

[Sim88] H. Simonis. Test Generation using the Constraint Logic Programming Language CHIP. In *Conference Proc. of Sixth International Conference on Logic Programming*, pages 155–160, August 1988.

[SKL90] Vijay A. Saraswat, Ken Kahn, and Jacob Levy. Janus: A step towards distributed constraint programming. In *Proceedings of the North American Conference on Logic Programming*, October 1990.

[SL92] Vijay A. Saraswat and Patrick Lincoln. Higher-order, linear concurrent constraint programming. Technical report, Xerox Palo Alto Research Center, August 1992.

[SR90] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages, San Fransisco*, January 1990.

[SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of Eighteenth ACM Symposium on Principles of Programming Languages, Orlando*, January 1991.

[Tse92] Clifford Tse. Linear Janus: A concurrent programming language. Technical report, Xerox PARC, 1992.

[Wad91] William Wadge. Higher-order Horn Logic Programming. In V. Saraswat and K. Ueda, editors, *Logic Programming (Proceedings of the 1991 International Symposium)*, pages 289–303. MIT Press, 1991.

[War87] D.H.D. Warren. The Andorra Principle. Presented at Gigalips workshop, U. of Bristol, 1987.