Actors as a Special Case of Concurrent Constraint Programming

Kenneth M. Kahn and Vijay A. Saraswat
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, CA 94304, USA
(415) 494-4390; 494-4334 (FAX)
Kahn@PARC.Xerox.com

This paper will appear in October 1990 in the Proceedings of OOPSLA/ECOOP90.

Keywords: Actors, Open Systems, Concurrent Programming, Constraint Programming

Abstract

Saraswat recently introduced the framework of concurrent constraint programming [14]. The essence of the framework is that computations consist of concurrent agents interacting by communicating constraints. Several concurrent constraint programming languages have been defined. They differ in the kinds of constraints that can be used as well as the kinds of operations on constraints which are available. In this paper we introduce a very simple concurrent constraint language we call **Lucy**, designed to closely mimic the actor model of computation. Agents can communicate only by the posting of constraints upon bags (unordered collections possibly with duplicate elements). This very impoverished concurrent constraint language is a syntactic subset of **Janus**, a concurrent constraint language which closely resembles concurrent logic programming languages such as Guarded Horn Clauses [21], Strand [5], Parlog [2] and Flat Concurrent Prolog [13]. By identifying the subset of **Janus** which is an actor language, we elucidate the relationship between actors and concurrent logic programming (and its generalization as concurrent constraint programming). **Lucy** is best not thought of as a unification of logic and constraint programming with actor and object-oriented programming, but as the *missing link* between these programming language genera.

1 Introduction

Actors are a minimal model of distributed computation. The model is lean and simple and adequately captures the essence of asynchronous message-passing systems. A good model for studying distributed systems need not, however, be a good basis for the design of programming languages intended to support the programming of large-scale distributed systems. We conjecture that the lack of wide-spread use of actor languages, despite twenty years of development, is due largely to the low-level nature of the actor model. We shall attempt to make the case that **Janus** as a superset of the **Lucy** actor language, enables the direct expression of many more common patterns of programming than actor languages while preserving the necessary properties for open-systems programming [10].

A major technical shortcoming of the actor model is that it is not directly compositional. A configuration of actors, in general, is not itself an actor since it can respond to messages arriving at different ports. Introducing the concepts of receptionist and configuration enables compositionality while increasing the complexity of the actor model [1]. A major advantage of concurrent constraint programming is that the parallel composition of agents is itself an agent. Unlike actors, an agent can be receptive to communications on any number of ports.

In the actor literature there is often talk of "mailboxes" or "mail addresses", though they are not part of the actor model. They are only *implicitly* part of an actor. This is because they are used in a standard manner which usually picks off and processes messages one by one. A major source of the expressiveness of **Janus** is that mailboxes are made *explicit*, agents may read from several of them, and separate read and write accesses may be communicated in messages. In order to mimic a simple **Janus** program in which agents use and communicate mailbox access rights, an actor program needs to build a relatively complex configuration of actors. The ability to read from multiple mailboxes also provides the basis for a simpler alternative mechanism for back communication than the continuations of actor computations. Multiple communication channels per agent and the communication of various kinds of access to communication channels are perhaps unusual programming language notions but are common in distributed operating systems (e.g. Mach [22]). As such, **Janus** models current practice in distributed computation more directly than the lower-level modeling provided by actors.

2 Distributed and Concurrent Constraint Programming

Concurrent constraint (cc) programming [14] is a synthesis and fundamental generalization of constraint logic programming (e.g. CLP(R) [8]) and concurrent logic programming (e.g. Flat Guarded Horn Clauses [21]). In the cc framework computation consists of a collection of agents interacting by performing various constraint operations upon shared variables. The most central operations are *ask* and *tell*. (This terminology was adapted from [12]. Agents may *tell* constraints thereby conjoining them to the pool of constraints that have hitherto been posted. The behavior of an agent is typically conditional upon the result of *asking* if this pool implies a given constraint, or its negation. If the pool implies neither the given constraint nor its negation, the agent is *blocked*; that is, it is inactive until other agents add constraints to the pool such that the pool has enough information to answer the given constraint. A *configuration* of the system, therefore, consists of a collection of *agents*, which collectively define the behaviors still to be executed, together with the pool of constraints that have been imposed hitherto.[1]

A language in the cc framework is obtained by choosing a particular ask-and-tell constraint system (which fixes the vocabulary of ask- and tell-constraints and their underlying domains of interpretation), and choosing a set of constraint operations (in addition to ask and tell the framework defines other operations, such as inform, check, initialize, etc. [14]). For example, the so-called concurrent logic programming languages such as Parlog, GHC and Atomic Herbrand are obtained by instantiating the Ask-and-Tell cc languages over the "Herbrand" constraint system, in which terms denote finite trees, and only equality constraints can be imposed. From an object-oriented perspective the cc languages offer a very rich framework in which entities may communicate over multiple 1–to–1, 1–to–many, many–to–1, and many–to–many channels a wide range of pieces of partial information, cooperating and competing in various ways, and possibly relying upon a sophisticated underlying inference mechanism. For example, an agent may tell the constraint $\texttt{Speed} > \texttt{140.3}$ thereby activating some other agent suspended on the ask "$\texttt{Car} = \texttt{porsche}$", because a complex of constraints present in the store may, together with the constraint $\texttt{Speed} > \texttt{140.3}$, entail that $\texttt{Car} = \texttt{porsche}$.

The languages discussed in this paper are very impoverished, relative to other members of the concurrent constraint programming family of languages. They support a limited vocabulary of constraints over only a few kinds of objects. This is because, like actors, they are intended for programming large-scale distributed systems. Issues of scalability, trust, reliability, and many more need to be taken into account [7, 10]. We identify the subset of cc programming languages which are suitable for distributed computing as the *distributed constraint* (**dc**) programming languages. A major characteristic of **dc** languages is the compile-time guarantee that the store of constraints cannot become inconsistent at run-time. This means that at run-time no constraint-solving is necessary—in particular this means that for **dc** languages which are defined over Herbrand-like constraint system (such as the language **Janus**), no unification is necessary at run-time. An inconsistent store implies *all* ask constraints,

[1] The pool of constraints is conceptually global, but is typically implemented in a distributed manner with only pairwise coordination.

as well as their negations, leading to widespread uncontrolled behavior. Instead of defining away the possibility of inconsistencies, languages like Guarded Horn Clauses [21] place responsibility on the programmer to keep the store consistent. In large-scale distributed systems this is unacceptable since the actions (whether malicious or accidental) of one component can bring chaos to all of the others. Another characteristic of the **dc** framework is that it must be possible to realize the constraint operations in a local manner.

One advantage of exploring new languages in the cc and **dc** frameworks is that much of the work has already been done in a general or parameterized fashion. Various combinators such as parallel composition or information hiding have already been formally defined. Formal semantics, proof methods, program transformations, etc. for large classes of cc and **dc** languages are active areas of research [17]. Exploring this space is a joy for a language designer who needs only concentrate on the pieces that are new rather than design a new language from scratch.

3 Actor Model of Computation

The actor model of computation is also a general framework for describing distributed computations. It is based upon the notion of asynchronous message-passing between encapsulated entities called actors. Upon receiving a message, an actor can create new actors, communicate by sending messages to other actors, and designate a replacement actor to service subsequent messages. An actor can only refer to its acquaintances, i.e., the set of actors it was created knowing about, those it learned about from messages it received, and those it created. It can only send messages to acquaintances and only create actors whose acquaintances are a subset of its acquaintances. More details can be found in [1, 3, 7].

4 Lucy – A DC Programming Language

We present here **Lucy**, a **dc** language, and show that it has the (informal) properties of an actor language.

The syntax used in this paper is a simple variation on the *clausal* syntax common to many concurrent constraint and logic programming languages, and used in [14]. A program is a collection of *clauses*, of the form:

```
behavior_name(ask_term, ..., ask_term) :: agent.
```

where an `agent` is, recursively, a tell-constraint, a simple agent (recursive invocation), a conditional agent (an ask-constraint, followed by an agent), or a parallel composition of agents:

```
agent :: =
      tell_constraint | name(tell_term, ..., tell_term) |
      ask_constraint → agent | agent, agent.
```

This syntax has a few advantages over the more conventional definite clauses. The choice of the syntax "$\rightarrow$" for "blocking ask" (conditional agents) is deliberate; this combinator can be construed as a logical entailment operator. It naturally and elegantly allows the specification of nested conditionals thus helping avoid the need for choosing arbitrary procedure names for every branch point in the program.

The operational semantics of **Janus** (and thereby **Lucy**) is easy to specify. A (non-terminal) configuration consists of a pair $\langle A, \sigma \rangle$, where A is an agent, and σ is the *store*, that is a pair of the form $c : V$, where c is a constraint, and V is the set of variables occurring in A and c. The "telling" of c in a store $c' : V$, terminates in one step, yielding the store $(c \wedge c') : V$. The execution steps of (A_1, A_2) are obtained by interleaving the execution steps of A_1 with those of A_2 in any order. The agent $(c \rightarrow A)$ can, in store σ reduce to the agent A provided that σ entails $\exists X_1, \ldots, X_k.c$ (where $X_1, \ldots, X_k$ are the "local" variables which do not occur in the head of the clause). In this case the store is augmented with an "answer constraint" on the local variables $X_1, \ldots, X_k$ which is conceptually generated by attempting to answer the "query" $\exists X_1, \ldots X_k.c$ given the constraint "database" σ. (Note that because of the presence of bags, there may be more than one answer to any such query. In such a case, any one answer is chosen arbitrarily.)

Similarly, in one step, the agent $p(t_1, \ldots, t_n)$ reduces in a store σ, if there is a clause in the program with head $p(a_1, \ldots, a_n)$ such that the query $\exists Y_1, \ldots Y_k.X_1 = a_1, \ldots, X_n = a_n$ can be answered from the store $\sigma \wedge X_1 = t_1 \ldots X_n = t_n$, where $Y_1, \ldots, Y_k$ are the "local" variables in the query. As for conditional agents, the store is augmented with an answer for the query, on the local variables. If there is enough information in the store to show that the goal cannot match any clause, then the goal is *removed*. [16] discusses why this is logically defensible—the essential point is to treat the bodies of clauses as certain kinds of implications rather than as conjunctions.

In **Lucy** the term $!X$ (called the *teller* for X) is used to indicate the right to tell constraints involving X. Legal **Lucy** (and **Janus**) clauses must satisfy certain syntactic restrictions designed to ensure that a behavior cannot use a $!X$ term to "read" (i.e. match on) the contents of X, and must use a $!X$ term when "writing" on (i.e. posting constraints on) X. The unadorned variable X is an ask right for X (called the *asker*). In **Lucy** for any given variable there are only two references – one the asker and the other the teller. The rules for ensuring this are presented in [15]. Variables denote point-to-point communication channels. Many-to-1 communication is accomplished using bags of askers.

The domain of the constraint system underlying **Lucy** consists of a set of constants ("urelements"; these include arithmetic constants) and (possibly infinite) unordered collections (bags) of such elements. Ask and Tell constraints are interpreted over this domain. The permissible ask constraints include:

- $\texttt{X} = \texttt{T}$ where T is any non-bag term.
- Various type and arithmetic tests such as $\texttt{X} \geq \texttt{Y}$, $\texttt{integer}(\texttt{X})$, etc.
- In addition, terms of the form $\{\texttt{M}\} \cup \texttt{B}$ may occur in the head of a clause, provided that `M` and `B` are variables that do not occur elsewhere in the head of the clause.

  Intuitively, an invocation can "match" a clause-head which contains the term $\{\texttt{M}\} \cup \texttt{B}$, if its corresponding argument is known to be a non-empty bag (for example, it may be a non-empty enumerated bag, or a variable `A` on which a constraint of the form $\texttt{A} = \texttt{X}_1 \cup \ldots \cup \texttt{X}_\texttt{n}$ has been imposed and at least one of the $\texttt{X}_\texttt{i}$ has been equated to a singleton). The match succeeds by equating `M` to one of the elements in the bag, and `B` to the "rest" of the bag. (See Appendix for formal details.)

The tell constraints permitted in **Lucy** are:

- $\texttt{X} = \{\texttt{T}\}$ where `T` can be a constant or a variable.
- $\texttt{B} = \texttt{X}_1 \cup \ldots \cup \texttt{X}_\texttt{n}$. This declares that `B` is the bag union of the $\texttt{X}_\texttt{i}$s. The $\texttt{X}_\texttt{i}$s are variables.

For convenience, we frequently write $\texttt{B} = \{\texttt{T}\} \cup \texttt{B}_1$ rather than $\texttt{B} = \texttt{B}_0 \cup \texttt{B1}$ and $\texttt{B}_0 = \{\texttt{T}\}$. In addition, arithmetic expressions and constants can occur in simple agents.

Lucy programs must follow the syntactic restrictions below:

1. *An agent asks about only one asker.* Simple agents can have read access to multiple bags but they can read from only one at a time. For example, the head of a clause cannot be $\texttt{name}(\{\texttt{M1}\} \cup \texttt{In1}, \{\texttt{M2}\} \cup \texttt{In2})$.
2. *No communication of askers.* If `T` in expressions of the form $\texttt{X} = \{\texttt{T}\}$ is a (unadorned) variable it must be inferable from the ask conditions of the clause that `T` is a constant or a teller.
3. *No internal choice.* An agent can have only one clause. The agent in the body of the clause can be conditional, of course. Each branch of a conditional has mutually exclusive ask conditions.

The first restriction is to match the actor notion that each actor always has exactly one mailbox it reads from. Tellers correspond to actor handles or mail addresses. Simple agents correspond to actors. An actor "becomes" (i.e. replaces itself by) the simple agent which is given the remainder of the bag of incoming messages. The second restriction is to fit the actor notion that mailboxes are implicit and cannot be communicated. The last requirement makes the internal behavior of agents deterministic; the only non-determinism in the program is the "arrival non-determinism" of messages, that is, the non-determinism involved in choosing one of possibly many messages in a bag.

The similarity between the conception of bags and actor mailboxes should now be clear. Roughly, asking $\mathtt{B} = \{\mathtt{M}\} \cup \mathtt{B1}$ is equivalent to removing the message `M` from `B` and naming the remainder `B1`. Unlike actors, however, **Lucy** does not permit multiple tellers and so an explicit operation is needed in order to split a bag (reference) into privately accessible pieces. For example, $\mathtt{X} = \mathtt{X}_1 \cup \mathtt{X}_2 \cup \mathtt{X}_3$ "splits" the reference to X into three references called $\mathtt{X}_1$, $\mathtt{X}_2$, and $\mathtt{X}_3$. The standard idiom to send a message `M` on a channel `C` and to name the channel for subsequent communications $\mathtt{C}_1$ is $\mathtt{C} = \{\mathtt{M}\} \cup \mathtt{C}_1$.

Lucy agents have the primitive ability to communicate only constants and tellers. Here we present how a **Lucy** agent can be defined which can represent structured messages. For simplicity, we consider a message agent to be capable of responding to two kinds of requests: "reply with your first component" and "reply with your second". More complex structures can be built in a similar manner. The program is non-trivial since the request itself needs to communicate both which component it is requesting and where to send the reply.

```
message({M} ∪ In, One, Two) ::
        % a message has arrived which is a teller,
        % acknowledge receipt,
        % and make Private be a new private channel
        M = !C →
          (C = !Private ∪ C1,
           message1(Private, !C1, In, One, Two)).

message1({M} ∪ FromClient, !ToClient, OriginalIn, One, Two) ::
        M = 1, constant(One) →
          (ToClient = {One},
           message(OriginalIn, One, Two)),
        M = 2, constant(Two) →
          (ToClient = {Two},
           message(OriginalIn, One, Two)),
        M = 1, One = !T →
          (T = T1 ∪ T2,
           ToClient = {!T1},
           message(OriginalIn, !T2, Two)),
        M = 2, Two = !T →
          (T = T1 ∪ T2,
           ToClient = {!T1},
```

```
        message(OriginalIn, One, !T2)).
```

After receiving a request a `message` agent creates another agent (`message1`) to service messages received in the original mailbox. `message1` is like an insensitive actor [1] in that, in addition to its own mailbox, it holds onto the original mailbox (`OriginalIn`). In the actor model an insensitive actor is implemented by a complex configuration of actors which buffer messages and wrap them with tokens indicating their source. In **Lucy** the analog of insensitive actors are agents with multiple askers only one of which is being "asked about" (and the others are there only to pass on to descendants). This enables a direct expression of the problem. Alternatively, we could have defined **Lucy** to restrict the number of askers per agent to one and added the ability to ask and tell that a constant or teller is "wrapped" with some token. This would result in clumsier programs but would adhere even closer to the traditional actor style of computation.

Note that `message` agents maintain state even though they correspond to pure (or "unserialized") actors. This is because in responding to a message they may have to split a reference, handing out one and retaining the other (as a replacement for the original reference).

We can now use these `message` agents to implement a simple bank account program presented below which accepts deposits, withdrawals, and balance queries:

```
account({M} ∪ In, Balance) ::
        M = !T → % received tell rights to a message agent
          (T = !FromM ∪ M1,
           % start to set up a private channel to M
           account1(FromM, !M1, In, Balance)).

account1({M} ∪ FromM, ToM, OriginalIn, Balance) ::
        % received channel, so ask for first component
        M = !T →
          (T = 1,
           account2(FromM, ToM, OriginalIn, Balance)).

account2({M} ∪ Ignore, !ToM, OriginalIn, Balance) ::
        % set up a fresh private channel to M
        ToM = !FromM ∪ Ignore,
        M = deposit →
          (deposit(FromM, OriginalIn, Balance),
           M = withdraw →
           withdraw(FromM, OriginalIn, Balance),
           M = current_balance →
```

```
    balance(FromM, OriginalIn, Balance)).

deposit({M} ∪ FromM, OriginalIn, Balance) ::
    % ask for the amount to deposit)
    M = !T →
    (T = 2,
     deposit1(FromM, OriginalIn, Balance)).

deposit1({X} ∪ Ignore, OriginalIn, Balance) ::
    X ≥ 0 →
      account(OriginalIn, Balance + X).

...% and similarly for withdraw and balance
```

A typical configuration might be

```
account(A, 100),
    A1 ∪ A2 = A,
    owner1(!A1),
    owner2(!A2).
```

`owner2`, for example, can deposit fifty dollars in the account by spawning $\texttt{message}(\texttt{M}, \texttt{deposit}, \texttt{50})$ and telling $\texttt{A2} = \{\texttt{!M}\} \cup \texttt{A2}'$.

4.1 Everything is an Actor

Most actor languages take the extreme stance that everything is an actor including numbers, messages, behaviors, etc.. A few such as Atolia [3], like **Lucy**, have values as well as actors.

We are currently exploring how **dc** languages like **Lucy** could also be designed so that primitive data types can always be interchangeable with instances of user defined data types which have the same behavior. We are considering a more primitive version of **Lucy** without constants but with the ability to ask to disjunctively wait on input from several bags. We believe that this "bare bones" **Lucy** is a general model of computation that, unlike pure actor languages which need "rock-bottom actors", need not have any primitive agents.

4.2 Fairness and Guaranteed Delivery

The actor model of computation is built upon an abstraction of "guaranteed delivery" of messages. Roughly if a message is sent, eventually it will be received. We feel that such a guarantee is not of much practical importance since it permits delays of millions of years. Consider the case where S has two clients C_1 and C_2. If C_1 is a non-terminating process continually sending messages to S, the actor model would require that if C_2 sent a message to S that it would eventually be processed. It is interesting to note that this cannot be truly guaranteed if C_1 and C_2 communicate to S via an Ethernet.

If guaranteed delivery is desired for **Lucy**, one could impose a similar restriction that if something is told to be a member of a bag and there is a sequence of asks splitting the bag into an element and the remainder, then splitting the remainder, and so on an unbounded number of times, that eventually the thing placed in the bag will be selected. It is not clear how such a fairness restriction could be expressed for the unrestricted **Lucy** language where agents may be receiving messages from multiple bags.

5 Janus – Lucy restrictions removed

The **Lucy** programs above are of course extremely clumsy. The programs are built from a very minimal foundation. Even low-level actor programs have built-in primitive actors for messages, continuations, and the like. From this crude binary message agent one could build up much more convenient kinds of messages. Rather than do that we consider the **dc** language **Janus** which is a superset of **Lucy**.

Janus removes the restrictions listed above thereby permitting agents to simultaneously read (either conjunctively or disjunctively) from multiple bags and to communicate askers as well as tellers. For convenience, **Janus** also permits tells of the form $\mathtt{X} = \{\mathtt{T_1}, \ldots, \mathtt{T_n}\}$ which sends n messages on the same channel. Similar restrictions limiting bag expressions to singletons are removed.

Bags provide a more general and less ad-hoc alternative to the primitive merge servers found in many other concurrent logic programming languages. Bags can support message sending in which messages are unordered as well as those in which the relative order of sends from each client is preserved. Unlike bags, primitive merges force servers to deal with lists of items and clients to follow a system-wide protocol for stream splitting. Rather than provide primitive implementations for a particular library agent, bags provide a more principled and general foundation for accomplishing the same goals.

Janus also introduces arrays which provide primitive support for structured messages. Not only do the programs become much more concise using arrays but the implementation is able to implement them in a conventional manner thereby bypassing a large number of message exchanges.[2] Arrays are constructed as terms of the form $< \mathtt{T_1}, \ldots, \mathtt{T_n} >$ and can be matched using the same syntax.[3]

[2] An interesting avenue for research is to explore whether compilers can be built which eliminate this message overhead.

[3] **Janus** has other array operations which support the same kinds of functionality and efficiency that arrays have in imperative programming languages while preserving the clean

The arrays of **Janus**, unlike agents which implement arrays (in a manner similar to the `message` agent), do not split references. Consequently arrays that may have tellers in them cannot be shared unlike the "sharing" of an agent via tellers to pieces of a bag.

Janus supports tells of the form $\mathtt{X} = \mathtt{Y}$ where `Y` can be an expression including a variable. This provides a very concise means of expressing transparent forwarding of messages. In **Lucy**, as in actor languages, one is forced, instead, to spawn forwarding agents to accomplish the same effect. This unnecessarily serializes the computation. Equivalence sets of variables are implemented without this serialization by **Janus** implementations; it is an open question as to whether such optimizations are feasible for actor programs relying instead upon transparent forwarders.

The ability to tell $\mathtt{X} = \mathtt{T}$ could cause problems if the constraint $\mathtt{X} \cup \mathtt{Y} = \mathtt{Z}$ had also been told, since `T` might be a constant. **Janus** avoids this problem by redefining "$X \cup Y$" to be the union of `X` and `Y` if they are bags and if either is a non-bag to treat it as the singleton bag containing that term. In **Lucy** this was not a problem since tells are of the form $\mathtt{X} = \{\mathtt{T}\}$.

Another extension in **Janus** is the introduction of the constant $\{\}$ to denote the empty bag. It has the property that $\mathtt{B} \cup \{\} = \mathtt{B}$. If clients explicitly drop their ability to communicate with a server by imposing the constraint $\mathtt{T} = \{\}$ then a client can detect when all of its clients have dropped their connection (since the bag it is reading is empty) and can therefore terminate. In **Lucy** and other actor languages a garbage collector is required to detect this and terminate the agent. Only for unreferenced cycles of **Janus** agents is a garbage collector needed.

Janus is a much more expressive language than **Lucy**. For example, the simple bank account can be written as:

```
account({M} ∪ In, Balance) ::
        M = <deposit, X>, X ≥ 0 →
          account(In, Balance + X),
        M = <withdraw, X, !Ack>, X ≤ Balance, X ≥ 0 →
          (Ack = ok, account(In, Balance − X)),
        M = <withdraw, X, !Ack>, X > Balance →
          (Ack = <overdraw_attempt, X>, account(In, Balance)),
        M = <current_balance, !Reply>, number(Balance) →
          (Reply = Balance, account(In, Balance)).
```

The guard `number(Balance)` was added to the last behavior of `account` to indicate that `Balance` is forced to be a constant and can therefore occur multiple

foundation as a constraint language. This extra functionality, however, is not used in this paper.

times. Straightforward flow-analysis-based compilation techniques can lead to the elimination of this test at run-time.

Many useful programming techniques are excluded by the actor/**Lucy** restrictions. For example, the bank account can be extended to use multiple readers to straight-forwardly program secure capabilities. Tellers to different bags being read by the same agent can correspond to capabilities or viewpoints. These idioms rely upon "disjunctively" waiting for messages from different sources. These techniques violate both the *an agent asks about only one asker* and *no internal choice* restrictions. The last requirement limits the source of non-determinism of **Lucy** to the arrival ordering of messages. **Janus** also has non-deterministic choice which is used to arbitrate between incoming requests on multiple ports. This yields the sort of non-determinism generated by Milner's "+" combinator (see [17]).

We illustrate this with an example. Suppose we wish to extend bank accounts to also receive input from bank administrators who are the only ones authorized to close accounts. We can give accounts ask rights to another bag which only bank administrators have tell rights to. The modified program includes the behavior:

```
account(In, {M} ∪ SysIn, Balance) ::
        M = <close_account> → closed_account(In).
```

where `closed_account` warns senders of messages that the account is closed. The original program is modified to pass through the extra parameter:

```
account({M} ∪ In, SysIn, Balance) ::
        M = <deposit, X>, X ≥ 0 →
          account(In, SysIn, Balance + X),
        ....
```

"Conjunctively" waiting for inputs from multiple sources before proceeding is also concisely expressible in **Janus**, as illustrated by the `match` agents in the queue server presented below.

```
queue(Input) ::
        split(Input, !Enq, !Deq), match(Enq, Deq).
split(<enqueue, X, Rest>, !En, De) ::
        En = <X, En1>, split(Rest, !En1, De).
split(<dequeue, !X, Rest>, En, !De) ::
        De = <!X, De1>, split(Rest, En, !De1).
match(<X, Enq>, <!Y, Deq>) ::
        Y = X, match(Enq, Deq).
```

The first clause defines `queue` agents, which will receive a sequence of `enqueue` and `dequeue` messages over time and will be able to respond as if these operations were being performed on a queue. Operationally, a `queue` agent splits into two agents. One monitors the input queue and distributes it into two queues, one consisting of the enqueue requests received (preserving order of arrival) and the other consisting of dequeue requests received. Each enqueue request contains the value to be enqueued. Each dequeue request contains a *teller* for a variable, the asker for which is retained by the agent sending the request. When an enqueue request arrives, it is communicated via the tell capability directly to the agent which requested that the dequeue operation be performed. (Note that the length of the dequeue list can be greater than the length of the enqueue list, so that the queue may be "negative" for a while.) The state of a positive length queue is the first argument to `match`. If the queue is negative its state is the second argument. If we wish to give clients separate "enqueue" and "dequeue" capabilities then only the `match` program is needed.

The ability in **Janus** to communicate askers enables configurations of agents doing either disjunctive or conjunctive waiting to be flexibly reconfigurable. We claim that these useful program idioms correspond to the construction of much more complex configurations of actors to accomplish the same ends.

Lucy corresponds to a low-level actor language like Act [1]. Several higher-level actor languages have been designed which provide a functional syntax that translates into a program using actor continuations. This is not particularly satisfactory since one either needs to give this higher-level language its own semantics (which may not look much like asynchronous message passing) or define it in terms of its translation to low-level actors. In the latter case, the problem is that the programmer while reasoning about and debugging their programs is forced to deal with the translation of his or her program to a form quite different from the original sources.

This situation should be contrasted with the relationship between **Lucy** and **Janus**. **Janus** is a much more expressive "higher-level" language which is a *superset* of **Lucy**. The benefits arising from the **Lucy** restrictions on **Janus** to make it fit the actor model are few. Most importantly, however, there is one uniform, simple, elegant and powerful way of viewing both **Lucy** and **Janus** computations: they involve the posting and checking of constraints by concurrently executing agents. This generalizes the fundamental notion of communication in concurrent logic programming languages, and provides a simple and elegant conceptual and semantic framework for analyzing these languages.

We take **Janus** to be representative of concurrent logic programming languages such as Strand [5] and Flat Guarded Horn Clauses [21]. Nearly all of the programming techniques commonly used in these languages carry over to **Janus**. In particular, nearly all of the object-oriented techniques and variants described in [11] are easy to reformulate in **Janus**. These programming techniques are expressible in **Janus** but not in **Lucy**. This extra expressiveness of **Janus**, was not obtained by sacrificing the "spirit" of actors — agents are still encapsulated, an agent can only communicate with its acquaintances, communication is asynchronous, etc. We have, however, introduced more primitive data types (e.g. arrays) in violation of the "everything should be an actor" principle.

Various **Janus** implementation efforts are underway. Hand compilations of simple programs indicate that **Janus** will run faster than Prolog and Strand. We hope to obtain speeds comparable to C. A networked version of **Janus** is also under construction. Since **Lucy** is just a syntactic subset of **Janus** then such implementations will also be **Lucy** implementations – modulo the syntax checker that verifies that the additional restrictions **Lucy** imposes are followed.

6 Related Work

In an influential paper [19], Shapiro and Takeuchi described how actors could be programmed in Concurrent Prolog. This inspired various attempts to make object-oriented languages based upon concurrent logic programming languages (e.g. [9], [6], [4]). The work reported here differs from these languages in many respects. First we have identified a subset of a concurrent constraint (logic) programming language which *is* an actor language – not one that can *implement* an actor language. In [19] many–to–1 communication, for example, was accomplished by networks of merge agents. Secondly we have explored the relationship between actors and a very simple and small language, based on asynchronous message-passing, and not on complex operations such as atomic unification of trees with "read-only" variables.

Various extensions to actor languages have been explored, some of which reify actor mailboxes (e.g. [20]). We are unaware, however, of any which support agents which can receive inputs on any number of "mailboxes" nor ones in which mailboxes can be communicated in messages.

In [11], one of the authors of this paper explores useful and significant variants of the basic object-oriented programming technique of Shapiro and Takeuchi. The paper presents examples illustrating the expressiveness that results from the ability to have multiple input channels per agents and the ability to communicate ask as well as tell rights. Equivalent actor programs are discussed and found to be much more complex and cumbersome. All but one of the examples presented there are easily written in **Janus** (the exception is the shared asker example for implicit multi-casting).

7 Summary

We have explored the relationship of the distributed constraint and actor frameworks by proposing a language which plays the role of a missing link between these frameworks. We presented **Lucy** a simple **dc** language which is an actor language. Our stance is that the actor model of distributed computation is a good one, but too low level for the basis of a programming language. **Lucy** is a syntactic subset of **Janus**, and we argued that those syntactic restrictions to fit the actor model are not well-motivated. Due to the **Janus** agents' ability to receive input from multiple ports, a configuration of **Janus** agents is itself an agent. This is not true of **Lucy** agent and actors. The restrictions placed upon **Lucy** eliminate very useful **Janus** programming techniques which rely upon multiple input ports and the ability to communicate input ports.

8 Acknowledgements

We are grateful for the fruitful discussions of **Lucy** we have had with Mark Miller, Volker Haarlsev, and Pat Hayes. The work on **Janus** is being done in collaboration with several other researchers, including Jacob Levy, Saumya Debray, Seif Haridi, Bogumil Hausman, and Mats Carlsson.

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems.* PhD thesis, University of Michigan, 1985.

[2] K. L. Clark and S. Gregory. Parlog: parallel programming in logic. *TOPLAS*, 8(1):1–49, January 1986.

[3] W.L. Clinger. *Foundations of Actor semantics.* PhD thesis, MIT, May. 1981.

[4] A. Davison. Polka: A Parlog object oriented language. Technical report, Department of Computing, Imperial College, London, 1988.

[5] Ian Foster and Stephen Taylor. Strand: A practical parallel programming language. In *Proceedings of the North American Logic Programming Conference*, 1989.

[6] K. Furukawa, Takeuchi. A., S. Kunifuji, H. Yasukawa, M. Ohki, and K. Ueda. Mandala: A logic based knowledge programming system. In ICOT, editor, *Proc of the Internation Conference on Fifth Generation Computer Systems*, 1984.

[7] Carl Hewitt. *The Ecology of Computation*, chapter Offices are open systems. Elsevier Science Publishers/North-Holland, 1988.

[8] Joxan Jaffar, J-.L. Lassez, and Michael Maher. *Logic Programming: Functions, Relations and Equations*, chapter Logic Programming Language Scheme. Prentice Hall, 1986.

[9] K. Kahn, E. Tribble, M. Miller, and D. Bobrow. *Research Directions in Object-Oriented Programming*, chapter Vulcan: Logical Concurrent Objects, pages 75–112. The MIT Press, 1987.

[10] Ken Kahn and Mark S. Miller. *The Ecology of Computation*, chapter Language Design and Open Systems. North Holland, 1988.

[11] Kenneth Kahn. Objects – a fresh look. In *Proceedings of the Third European Conference on Object-Oriented Programming*, pages 207–224. Cambridge University Press, July 1989.

[12] Hector J. Levesque. Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23:155–212, 1984.

[13] Colin Mierowsky. Design and implementation of Flat Concurrent Prolog. Technical Report CS84-21, Weizmann Institute of Science, December 1984.

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

[15] Vijay A. Saraswat, Kenneth Kahn, and Jacob Levy. Janus—A step towards distributed constraint programming. North American Logic Programming Conference, October 1990.

[16] Vijay A. Saraswat, Kenneth M. Kahn, and Jacob Levy. Distributed constraint programming—the dc framework and Janus. Technical report, Xerox PARC, August 1989.

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

[18] Vijay A. Saraswat, Martin Rinard, and Prakash Panagaden. Fully abstract "may" semantics for concurrent constraint languages. Technical report, Xerox PARC, March 1990.

[19] Ehud Shapiro and A. Takeuchi. Object oriented programming in concurrent prolog. *New Generation Computing*, 1:25–48, 1983.

[20] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with enabled-sets. In *Proceedings of Conference on Object-Oriented Programing: Systems, Languages and Applications, New Orleans, Louisiana*, pages 103–112, October 1989.

[21] K Ueda. Guarded Horn Clauses. Technical Report TR-103, ICOT, June 1985.

[22] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and D. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh Symposium on Operating System Principles*, November 1987.

A Operational semantics

The following discussion is based primarily on [14], which presents the semantics of the general framework, though it does not, of course, discuss the particular constraint system underlying **Lucy**. A bisimulation semantics for the cc languages is given in [17] and a denotational semantics fully abstract with respect to a notion of "may" observations is given in [18].

The following definitions will be useful in what follows. Let $g \equiv p(a_1, \ldots, a_n)$ and $g' \equiv q(t_1, \ldots, t_m)$ be two atomic formulas. Then by $g = g'$ we mean the constraint $(p = q, m = n, a_1 = t_1, \ldots, a_i = t_i)$, for $i = min(m, n)$. Further, for any syntactic entity ϕ, by $\mathbf{var}(\phi)$ we denote the (finite) set of variables that occur in ϕ, and by $\delta V.\phi$ we denote the formula $\exists X_1, \ldots, X_n.\phi$, where $\{X_1, \ldots, X_n\} = \mathbf{var}(\phi) \setminus V$.

Let $K \equiv g :: A$ be a program clause, and let $V = \mathbf{var}(g)$. Let K' be the clause obtained from K by replacing each sub-agent $c \rightarrow A'$ in A by $(\delta V.c) \rightarrow (c, A)$. For P a set of program clauses, define

$$[P] \stackrel{def}{=} \{K' \mid K \text{ is a variant of a clause in } P\}$$

The operational semantics of **Lucy** programs is defined — as for all cc languages — by means of a binary transition relation on a set of configurations Γ. In the following, let σ range over the set of *stores*, that is, pairs of the form $c : V$, where V is a set of variables (the variables underlying the store), and c is a constraint on V. A *configuration* is either a store (such a configuration is terminal), or a pair of the form $\langle A, \sigma\rangle$, where A is an agent, and σ a *store*. If $\sigma \equiv c : V$, we say that $\sigma \rightarrow c'$ iff $c \rightarrow c'$. Also, we let θ range over the set of idempotent substitutions, that is, mappings from the set of variables to the set of terms which are equal to the identity mapping almost everywhere, and which satisfy the condition that $\theta(\theta(X)) = \theta(X)$, for any variable X. (We assume that θ is extended to a mapping from terms to terms in the usual way.)

Let P be a **Lucy** program. The binary transition relation $\longrightarrow \subseteq \Gamma \times \Gamma$ is defined to be the smallest relation which satisfies the axioms given below.

The simplest axiom has to do with the effect of a tell operation:

$$\langle c, c' : V\rangle \longrightarrow (c \wedge c') : V \tag{1}$$

To tell a constraint is merely to add it to the store.

Consider next the reduction of a simple agent. Let $K \equiv (g :: A) \in [P]$ be a clause such that $K \cap V = \emptyset$. Let θ be a substitution such that $\mathcal{C} \models (c \rightarrow (g' = \theta(g)))$. Then we have:

$$\langle g', c : V\rangle \longrightarrow \langle \theta(A), (c \wedge g' = \theta(g)) : (V \cup \mathbf{var}(K))\rangle \tag{2}$$

Thus a simple agent can reduce with a clause provided that it satisfies the ask conditions associated with the clause.

The operational semantics of conditional agents is straightforward:

$$\frac{\mathcal{C} \models \sigma \to c}{\langle c \to A, \sigma\rangle \longrightarrow \langle A, \sigma\rangle} \quad \frac{\mathcal{C} \not\models \sigma \to c}{\langle c \to A, \sigma\rangle \longrightarrow \sigma} \tag{3}$$

Finally, parallel composition of two agents just means that their basic atomic steps are interleaved:

$$\frac{\langle A_1, \sigma\rangle \longrightarrow \langle A_1', \sigma'\rangle \mid \sigma'}{\begin{array}{c}\langle (A_1, A_2), \sigma\rangle \longrightarrow \langle (A_1', A_2), \sigma')\rangle \mid \langle A_2, \sigma'\rangle \\ \langle (A_2, A_1), \sigma\rangle \longrightarrow \langle (A_2, A_1'), \sigma')\rangle \mid \langle A_2, \sigma'\rangle\end{array}} \tag{4}$$