In Proc. 12th Symp. on Comp. Arch. and High Perf. Computing 20–37, São Pedro, SP, Brazil. October 2000.

Clara: An Actor Language for High Performance Distributed Computing

Carlos H. C. Duarte[1], Carolyn L. Talcott[2]

[1] BNDES, Av. Chile 100
Rio de Janeiro, RJ, Brazil, 20001 970
{carlos.duarte@computer.org}
[2] Stanford University, Gates Building
Stanford, CA, USA, 94305 2140
{clt@cs.stanford.edu}

Abstract — **In this paper we present Clara, a new programming language for high performance distributed computing. Clara has been developed to embody in an efficient distributed computing environment the conceptual clarity of the actor model, an object-based framework for the design and implementation of open distributed systems. We describe our Clara compiler, paying special attention not only to the adopted two stage translation process but also to the runtime environment, which is based on the message passing interface standard.**

Keywords —**Actor model, Object-based languages, Message passing interface, Compiler construction, Distributed systems.**

I. Introduction

With the proliferation of diverse operating environments and the intense use of software development tools in the area of compiler construction, the landscape of programming languages for distributed computing appears to have reached a point of saturation. Many such languages do exist, but the problem of developing from principles conceptually clear efficient distributed systems seems to be far from a solution.

In part, the problem above can be attributed to the lack of criteria in the development of distributed programming languages. Most existing languages do not follow an underlying model and simply result from the assembly of disconnected features. Another cause seems to be the lack of reuse in the design and implementation of distributed computing environments: they are developed from scratch, sometimes using the same production tools, but best performance practices learnt in a project are seldom replicated.

In this paper we present Clara, a new programming language for high performance distributed computing. Clara is based on the actor model of distributed systems [Agh86]. Actors are independent encapsulated units of control and computation. They interact solely via point-to-point asynchronous message passing. The delivery of messages in the actor model is guaranteed and, as a result of consuming a message, an actor may change its local state, create finitely many new objects or dispatch a finite number of messages to its acquaintances, the actors whose address was received at creation time or in a message. Based on this simple model, we believe that Clara is a conceptually clear language.

The paper discusses the development of a Clara compiler with a distributed runtime environment. The syntax and semantics of Clara are based on the high level actor language studied in [MT99], with the definition of functions particularised to use the purely functional style of the language Miranda [Tur86]. The runtime environment is built on top of LAM [BDV94], a distributed middleware which follows the standardisation efforts of the Message Passing Interface Forum [For97]. The compilation process takes as input high level Clara programs, strips out synchronous communication primitives which are not supported by the actor model (but are included here to ensure programmer comfort) and produces in the end a C program that makes use of LAM system calls. Reusing this publically available toolbox, we obtain a high performance tool that can be used in many distributed operating environments.

We organise the remainder of the paper as follows. Section II contains the syntactic and semantic definition of Clara. In Section III, an example of the use of our tool is presented. Our Clara compiler and runtime environment are outlined in Section IV.

II. The Syntax and Semantics of Clara

Clara is based on the actor model of distributed systems. Each actor has a finite set of attributes, state variables which may only be changed by local computations. Since interaction is via asynchronous (non-blocking) message passing, in a system of actors there is no notion of global state. Communities of actors having an explicit interface with their environment — the names of actors which may receive messages from the outside word (the receptionists) and those assumed to exist externally (the externals) — are regarded as components and define units of modularisation and coordination.

The abstract syntax of Clara is presented in the Appendix. There are just two kinds of data types in our language: actor addresses and integer numbers. The domain of actor addresses is endowed with a bottom element denoting undefined objects, represented by `nil`. Addresses provide an un-

ambiguous location transparent way of making references to actors, which are in fact structured objects. Each actor knows its own address (or name), which is recovered using the function `self`. Regarding integer numbers, all the arithmetic operations are supported. Clearly, we could have decided to support other more complex data types as a built in part of Clara, but these are definable in our language as illustrated in Section III. We assume as given a countably infinite set of variables, over which we form expressions and conditions in the usual way.

Each actor has a set of attributes and can accept messages of fixed types. Each message has a type tag and may carry a list of arguments, the message body. Methods define how messages are consumed. The message associated with the execution of a method may specify a customer, which is recovered using the function `cust`. The first method that matches the tag and pattern of arguments in an incoming message determines the commands to be performed as a response, a method body. Local synchronisation constraints, `disabledby` c where c is a condition, can constrain the intervals of time when the message may be consumed and are written in terms of the message body and the actor attributes. Syntactic descriptions of actors based on these features are called behaviour definitions.

Method bodies describe how an actor reacts to the reception of messages. Reactions are specified as sequences of commands, which may contain an attribute change (an expression assigned to the attribute name), the dispatch of a message (`send` followed by an expression producing the target actor name, by a message tag, a list of argument expressions and possibly another expression overwriting `nil` as the customer object), a reply to a blocking invocation of a remote method (`reply` followed by a reply value expression), a do nothing, conditional or block construction command (`skip`, `if then else` and `begin end`, respectively). Note that an iteration command is not present, but can be simulated through the self dispatch of a continuously and uniquely enabled message until a termination condition is fulfilled.

Commands are constructed out of expressions, which may be purely functional or denote an actor operation. Functional expressions are written using patterns (constants, variables, attributes, `self` and `cust`), arithmetic operations and function calls. Here functions are specified as in the side effect free purely functional language Miranda, through a mutually recursive list of equations. In each equation, the function name is followed by a list of patterns, which is succeeded by a list of guarded expressions. The first equation whose pattern list matches a function call and is followed by an expression whose guard is true determines how the function result is computed. In turn, actor expressions can represent a new object creation, specified as `new` followed by a behaviour definition name and a list of expressions that determine unspecified initial attribute values. Each creation results in a fresh actor name. The other type of actor expression is a remote method invocation with an rpc like semantics: `call` followed by an expression producing the target actor name, by a message tag and a list of call argument expressions. Each invocation produces the method return value, `nil` if not specified. It is important to mention that, because actor names are a basic data type, they may be referenced in the scope of functions, but actor expressions that either result in a side effect — creation or synchronous invocation — or need an evaluation context — `self`, `cust` and attributes — may only appear in a behaviour definition body.

Components are the units of modularisation in the definition of programs. They have interfaces, which are defined as pairs of lists of variables specifying the initial externals and receptionists of each component. Components also have internal actors, which are super sets of the receptionists. The initially present internal actors and the messages addressed to these objects are specified in the component definition. These may be statically or dynamically composed, but the description of composition operations is omitted here.

The definitions of functions, behaviours and components are uniquely identified. These definitions are packed into libraries, which are one type of compilation unit. The other type is that of programs, which result from the application of composition operations, including the identity, over component names and possibly interfaces. Each definition specifies the scope of identifiers and their binding to data objects. For instance, an attribute name may only appear within the enclosing behaviour definition and a data object represented by a variable in a function argument may only be referenced in this way within the function definition body.

Since Clara is based on the user language studied in [MT99], it inherits both formal semantics specified therein. The high level semantics assumes that each object is single threaded. The low level formalism is in terms of fair transition systems where each configuration is determined by the set of existing components at each stage, including their actors and messages in transit, and transitions between configurations are defined by reaction rules of the constituent components: idle or local computation, creation of new actor and the dispatch or delivery of a message. The formal semantics of composition is defined in [Tal98].

III. Distributed Computing with Clara: An Example

Clara is a language particularly well suited to the implementation of open distributed systems. In this section we exemplify this through the computation of spanning trees for dynamically configurable objects organised as a network. This construction can be used to simulate another mode of interaction: multicasting of messages to actor groups.

The basic structure of a multicasting system contains two components, the networked objects and the spanning tree.

```
component Network
receptionists  [nr]
actors         nr := new NNode[];

component Tree
externals     [nr,tp]
receptionists [tr]
actors        tr := new TNode[nr,tr,tp];
messages      send tr BSYNC[];

component System
 Network([],[nr\network])||
 Tree([nr\network,tp\par],[tr\tree])
 ([par],[tree,network])
```

Fig. 1. Basic definition of multicast system components.

The initial definition of these components appears in Fig. 1. At first, the network contains only one object, nr, which may receive messages from the environment. The tree component contains just one receptionist actor as well, the root node tr. The latter component regards as externals not only the represented networked object but also a parent actor for the spanning tree, tp, which may be useful as an extension point in case the network becomes part of a larger one. There is also a pending BSYNC message addressed to the tree root which, once consumed, will start up the monitoring of connections of the object network. These definitions could be statically composed as presented in the figure.

The definition of our system, System, says that the object whose existence in the environment was assumed by the tree definition is bound to the initially existing networked object. Moreover, the resulting structure has two receptionists and one external.

The behaviour of networked objects and spanning tree nodes is presented in Fig. 2. The network has a dynamic topology which may be expanded in two ways: creating new nodes, in which case a ADD message is dispatched to the actor responsible for the new object creation, or making a connection to another network, in which case a LINK message containing the network address is dispatched to the same object. As a simplification, we allow each networked object to maintain only two directed connections. The tree root is supposed to receive each message to be multicast, MSG. This object maintains a representation of the network which is a directed acyclic connected graph without converging links. This structure is used to guarantee that message multicast, which is synchronous within the tree to avoid deliveries to unintended recipients, be performed with minimal overhead for the networked objects themselves.

It can be noted in the definition of network nodes, NNode, that even after having processed the addition of a new node or the connection to another network, the respective objects may answer queries concerning the established connections, ASK, without reporting yet the existence of newly introduced links. This happens because a self addressed asynchronous message, SET, is dispatched and only upon its processing will the links be updated properly. This may happen after the arrival of the query, since there is no requirement on the order of message arrival in the actor model. To decrease the nondeterminism in the specified behaviours, the asynchronous use of SET may be substituted by a synchronous invocation, using call instead of send. This illustrates the importance of supporting both synchronous and asynchronous self invocations, to avoid duplication of code providing the same functionality and facilitate reuse.

Once the initial BSYNC message is processed by the spanning tree root, this component starts to traverse the object network gathering information about existing links, using ASK, and checking if they have already been represented as tree nodes, via INS. The recursion of this process happens with the dispatch of other BSYNC messages to the children of each node, if they exist. When some node receives a ANS message containing a link, which is presumed as yet to be represented, the whole tree is searched to verify if it is not already represented somewhere. For each visited node with a child, a join-continuation object with behaviour Cont is created to wait for either one INSERR message, stating that the link is already in the tree, or two of type INSOK, saying that the search was not successfull. In the end, a similar message is dispatched to the query originator, which in turn decides on whether or not to create a representation of the new link connected to itself.

The process above would work as expected if the object network were static, but in an unconstrained system objects could be created or new networks connected while the tree is being updated, leading to a behaviour which could never result in a network representation. We solve this problem assuming the existence of a wrapper to sit between System and any customer. This new component allows, in a mutually exclusive way, the tree to produce an up to date representation of the network and this last component to behave as described above, also hiding networked objects from the outside world. In this way, the illusion of a single object is created for the group of networked objects, as required in strict multicasting systems.

The definition of wrappers appears in Fig. 3. There, a typical use of synchronisation constraints is presented. Any wrapper object may be in one of two states: waiting for the tree to synchronise with the network ($s = 1$) or allowing this last component to receive environment requests ($s = 0$). In this second state, environment messages are recast and dispatched to the network. We need in this process helper objects filtering replies to the environment and avoiding that network addresses become known.

```
behaviour NNode
attributes n:=0; nb1:=nil; nb2:=nil;
methods
SET x: if (nb1==nil) then
      nb1:=x;
   else
      nb2:=x;
ADD c: if (n==2) then
      send c AERR[self];
   else begin n:=n+1;
      send self SET[new NNode[]];
      send c AOK[self];
   end;
LINK x c: if (n==2) then
      send c LERR[x,self];
   else begin n:=n+1;
      send self SET[x];
      send c LOK[x,self];
   end;
ASK c: send c ANS[self,nb1,nb2];
MSG: skip; /* not specified */

function
Used x y = 0, (x==nil) /\ (y==nil)
         = 1, (x==nil) \/ (y==nil)
         = 2, otherwise

behaviour Cont
attributes o; m; r:=0;
methods
INSERR:
   if (r<m) then send o INSERR[];
   r:=m;
INSOK x b: r:=r+1;
   if (r==m) then send o INSOK[x,b];
```

```
behaviour TNode
attributes n; r; p;
   s:=0; v:=0; aux:=nil; nc1:=nil; nc2:=nil;
methods
MSG: v:=call n MSG [];
   if ~(nc1==nil) then v:=call nc1 MSG[];
   if ~(nc2==nil) then v:=call nc2 MSG[];
BSYNC: s:=1; send n ASK[self];
ESYNC x: s:=s+1;
   if (s > Used(nc1,nc2)) then
   begin s:=0; send p ESYNC[self]; end;
ANS n x y: if (Used(x,y)==0) then
      send self ESYNC[self];
   else begin
      if ~(nc1==nil) then send nc1 BSYNC[];
      else if ~(x==nil) then
         send r INS[self,x,0];
      if ~(nc2==nil) then send nc2 BSYNC[];
      else if ~(y==nil) /\ ~(y==x) then
         send r INS[self,y,1];
   end;
INS o x b: if (x==n) then
      send o INSERR[];
   else if (Used(nc1,nc2)==0) then
      send o INSOK[x,b];
   else begin
      aux := new Cont[o,Used(nc1,nc2)];
      if ~(nc1==nil) then
         send nc1 INS[aux,x,b];
      if ~(nc2==nil) then
         send nc2 INS[aux,x,b];
   end;
INSERR: send self ESYNC[self];
INSOK x b: aux := new TNode[x,r,self];
   if (b==0) then nc1:=aux;
   else nc2:=aux;
   send aux BSYNC[];
```

Fig. 2. Behaviour definitions of basic objects in the multicasting system.

IV. On the Translation and Execution of Clara Programs

The compilation of Clara programs is quite standard. Each program is passed as input to the compiler front end, which generates a syntax tree. This structure is analysed by a pre-compiler, which strips out synchronous construct nodes and produces a modified tree. The code generator takes the resulting structure and creates a C program. In the end, we need a C compiler to produce an executable.

The compiler front end was obtained from a specification of the language syntax, written in augmented BNF, using the standard compiler construction tools Lex and Yacc. These tools were used to generate a code fragment, written in C, that produces, when executed having a source program as input, the corresponding annotated syntax tree. The annotations are used in the generation of code. Incidentally, the whole development of this component was greatly simplified due to the reuse, as part of our implementation, of the Miranda syntax specification described in [DI96].

The next stage in the compilation process takes the generated syntax tree and removes in a recursive traversing any use of synchronous interaction constructs: synchronisation constraints and remote method invocations. These are identified by tree node annotations. Actually, the translation begins by rewriting expressions, conditions, and then commands, methods, behaviour and component definitions, but to ease the presentation we organise the material according to the two constructs above. In the sequel, we just sketch the translation schema π, first treating the removal of synchronisation constraints (`disabledby`) and then treating the elimination of remote method invocations (`call/reply`).

A. Compiling Synchronisation Constraints

The translation of synchronisation constraints is quite subtle, since it cannot violate the fairness requirement on the delivery of messages present in the actor model. Mason and Talcott [MT99] note that the simple solution of translating the arrival of a disabled message into its self dispatch does

```
behaviour Wrapper
attributes n; t; s:=1; v:=0;
methods
BSYNC disabledby (s==1):
   send t BSYNC[]; s:=1;
ESYNC t disabledby (s==0):
   send self BSYNC[]; s:=0;
MSG disabledby (s==1):
   v:=call t MSG [];
ADD c disabledby (s==1):
   send n ADD[new Helper[self,c,nil]];
LINK x c disabledby (s==1):
   send n LINK[new Helper[self,c,x]];

behaviour Helper
attributes w; c; t;
methods AOK x: if ~(c==nil) then
   begin send c AOK[w]; c:=nil; end;
LOK x n: if ~(c==nil) then
   begin send c LOK[x,y]; c:=nil; end;
AERR n: send n ASK[self];
LERR n: send n ASK[self];
ANS n x y: if (Used(x,y)==2) then
   if (t==nil) then send x ADD[self];
   else send y LINK[x,self];

component Wrap
externals     [network,tree]
receptionists [wrap]
actors wrap := new Wrapper[tree,network];

component Group
 System([par\wrap],[tree,network])||
 Wrapper([tree,network],[wrap]))
 ([],[wrap])
```

Fig. 3. Definition of wrappers and related structures.

not provide the intended level of fairness since the message could be infinitely often enabled but never processed because it always arrives when disabled. We follow their strategy providing a fair solution which relies on a queue of delivered messages with checked and unchecked status. The behaviour of an actor system obeying this refined protocol is presented in Fig. 4, where state transitions are labelled with condition | effect pairs.

Let us sketch the compilation of synchronisation constraints. We consider that any actor specified in a high level program is realised as two low level objects, a message queue actor and a behaviour actor. This system cycles between two states, represented by the value of the attribute t: incoming messages may be processed ($t = 0$); or the system is traversing a delivered message queue searching for enabled messages ($t = 1$), in which case it cannot deal with fresh messages that are instead stored in a pending message queue. We assume the existence of a queue object, q, whose behaviour depends on t and the status s of stored messages: the received message is unchecked ($s = 0$); the checked message was enabled ($s = 1$); the message was checked and consumed ($s = 2$); the checked message was disabled ($s = 3$). We also consider the existence of a behaviour actor, which, differently from [MT99], receives from the queue messages annotated with their respective status, customer and the queue address. After receiving a checked enabled message, the queue actor behaves as the given object.

The specified creation of an actor in a high level program is translated as follows, where a queue is defined to serve as a wrapper for the requests addressed to the specified object:

$$\texttt{new Source}[e_1, \ldots, e_n]$$

$$\pi \Downarrow (1)$$

$$\texttt{new } Q^{\texttt{Source}}\texttt{[new Source}[\pi(e_1), \ldots, \pi(e_n)]]$$

According to Fig. 4, the initial state of an actor system corresponding to a high level object is one willing to deal with unchecked messages. Whenever a new message is received, it is immediately stored in the queue. The queue dispatches to the behaviour actor a copy of each unchecked message, which may be fresh or come from the pending message queue. Suppose the message matches a method defined by:

$$MT^j\, x_1^j \ldots x_m^j \texttt{ disabledby } c^j : b^j$$

The behaviour object may find out that the message is currently disabled (c^j true), in which case the process of traversing the queue continues with a new message being requested, NX_MT^j[1]. Otherwise, a request to mark the received message as enabled is issued to the queue, EN_MT^j. This last request causes a dispatch to the behaviour actor of another copy of the message with an updated status, which generates, once consumed, yet another request for the queue to mark the message as consumed, CK_MT^j. In case the behaviour actor receives a checked message with disabled or processed status, this indicates that the end of queue was reached and the messages which have not been consumed should be considered as unchecked. In the end, there may still exist pending unchecked messages, which are then checked, or the system returns to the state in which environment messages are awaited. We adopt the following schema to capture this behaviour in processing a MT^j message:

$$\begin{array}{l}
\texttt{behaviour Source} \\
\texttt{attributes } a_1; \ldots; a_i; \\
MT^1\ x_1^1 \ldots x_l^1 \texttt{ disabledby } c^1 : b^1; \\
\ldots \\
\hline
\ldots \\
MT^j\ x_1^j \ldots x_m^j \texttt{ disabledby } c^j : b^j; \\
\ldots \\
\hline
\ldots \\
MT^k\ x_1^k \ldots x_n^k \texttt{ disabledby } c^k : b^k;
\end{array} \quad \overset{\pi(2)}{\Rightarrow}$$

(*) s^j, k^j, q^j fresh identifiers.

[1] Since the symbol _ is not allowed in high level programs, message tags formed like this one are necessarily fresh.

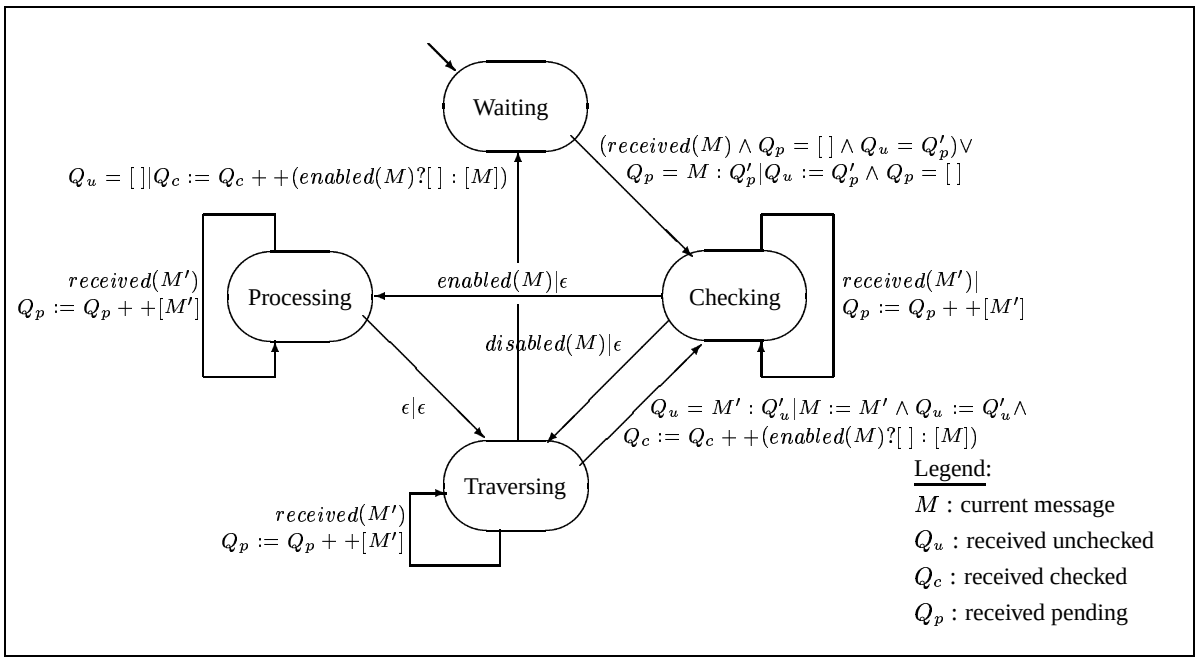

Fig. 4. States of an actor system simulating synchronisation constraints using message queues.

```
behaviour Source
attributes a_1;...;a_i;
MT^1 x_1^1 ... x_l^1 disabledby c^1 : b^1;
...
MT^j x_1^j ... x_m^j s^j k^j q^j: if (s^j==0) then
   if π(c^j) then
      send q^j NX_MT^j[x_1^j,...,x_m^j,k^j,self];
   else
      send q^j EN_MT^j[x_1^j,...,x_m^j,k^j,self];
 else
   if (s^j==1) then begin
      π(b^j); send q^j CK_MT^j[x_1^j,...,x_m^j,k^j,self];
   end; else send q^j _UNCK[self];
...
MT^k x_1^k ... x_n^k disabledby c^k : b^k;
```

Objects with a Q^{Source} behaviour are specified below. They accept messages as the original actor, o. Each received message of type MT^j is stored as a new $\overline{MT}^j$ object with the unchecked status. After this new object creation, if the queue has just entered into the traversing state, a request for the dispatch of a copy of the new unchecked message to the behaviour actor is issued and the reference to the queue element, u, is updated. The queue object may already be in a travesing state, in which case the new message is just stored in the pending queue, which begins with p. Note that we use this representation to implement message queues as chains of connected actors. From the dispatch of the new request onwards, the queue and behaviour actors engage in the communication protocol above. The queue object serves as a gateway for any access of the behaviour actor to queued messages, forwarding requests to the queued object which was the last one before the beginning of the traversing process, u. The use of o and u only as part of internally exchanged message bodies garantees that the respective objects are hidden within the system, since their names do not become known by the environment. We present below the definition of these objects, which is included in the compiled code:

```
behaviour Q^Source
attributes  o; t:=0;
            p:=nil; u:=nil; n:=nil;
...
MT^j x_1 ... x_m : n:= new MT^j_Source
     [x_1,...,x_m,0,cust,self,p,o];
  if (t==0) then begin
    t:=1; u:=n; send u _NEXT[];
  end; p:=n;
EN_MT^j x_1 ... x_m x_k x_o : if (x_o == o) then
  send u EN_MT^j[x_1,...,x_m,x_k];
NX_MT^j x_1 ... x_m x_k x_o : if (x_o == o) then
  send u NX_MT^j[x_1,...,x_m,x_k];
CK_MT^j x_1 ... x_m k x_o : if (x_o == o) then
  send u CK_MT^j[x_1,...,x_m,x_k];
...
_UNCK x_o: if (x_o == o) then
    send u _UNCK[u];
_END x_u : if (x_u == u) then begin
  if (p==u /\ ~(u==nil)) then
    t:=0;
  else if ~(u==nil) then begin
    u:=p; send u _NEXT[];
  end;
end;
```

```
behaviour MT^j_Source
attributes a_1; ...; a_m; s; k; q; n; o;
_NEXT : if (s==0 \/ n==nil) then
    send o MT^j[a_1, ..., a_m, s, k, q];
  else send n _NEXT[ ];
_UNCK x_u : if (s==3) then s:=0;
  if (n==nil) then send x_u _END[ ];
  else send n _UNCK[x_u];
_END : send q _END[self];
...
EN_MT^j x_1 ... x_m x_k : /* 1 ≤ i ≤ m */
  if (a_i==x_i)/\(k==x_k)/\(s==0) then begin
    s:=1; send o MT^j[x_1, ..., x_m, s, x_k, q];
  end; else send n EN_MT^j[x_1, ..., x_m, x_k];
CK_MT^j x_1 ... x_m x_k : /* 1 ≤ i ≤ m */
  if (a_i==x_i)/\(k==x_k)/\(s==0) then begin
    s:=2; send self _NEXT[ ];
  end; else send n CK_MT^j[x_1, ..., x_m, x_k];
NX_MT^j : /* similar to CK_MT, with s:=3 */
...
```

B. *Compiling Remote Method Invocations*

Once the behaviour actor receives a message with checked enabled status, it will only expect to receive other ones from the queue after having replied with a confirmation that processing the current message was completed, CK_MT^j. We take advantage of this fact to translate each remote method invocation, which requires precisely a blocked state of the source actor until a reply is received from the target indicating that the method was completed. Suppose we are dealing with an actor with queue q^j and with method MT^j having customer k^j, some of the message arguments introduced due to schema (2) above. In this context, we perform the following translations, assuming the existence of new object attributes rb and rv to deal with the existence of a reply and the respective future value:

$$\begin{array}{rcl} \texttt{self} & \overset{\pi(3)}{\Rightarrow} & q^j \\ \texttt{cust} & \overset{\pi(4)}{\Rightarrow} & k^j \\ \texttt{reply}\ e & \overset{\pi(5)}{\Rightarrow} & \texttt{if}\ (rb == 0)\ \texttt{then} \\ & & \texttt{begin}\ rv := \pi(e); rb := 1;\ \texttt{end} \end{array}$$

Any method may be invoked in a remote synchronous way. The correct behaviour of an actor providing a MT^j method synchronously can be ensured by the translation presented below, where the return of the first reply value to the method customer is added as the last method body command:

```
behaviour Target
attributes a_1; ...; a_i;
MT^1 x_1^1 ... x_l^1 s^1 k^1 q^1: ... b^1; ...
...
---------------------------------------------
...
MT^j x_1^j ... x_m^j s^j k^j q^j: ... b^j ...;
...
---------------------------------------------
...
MT^k x_1^k ... x_n^k s^k k^k q^k: ... b^k ...
```

$\pi \Downarrow (6)$

```
behaviour Target
attributes a_1; ...; a_i; rb := 0; rv := nil;
MT^1 x_1^1 ... x_l^1 s^1 k^1 q^1: ... b^1; ...
...
---------------------------------------------
...
MT^j x_1^j ... x_m^j s^j k^j q^j: ... b^j; ...
  if ~(k^j==nil) /\ rb == 1 then
    send k^j _[rv];
  rb := 0;
...
---------------------------------------------
...
MT^k x_1^k ... x_n^k s^k k^k q^k: ... b^k; ...
```

In the compilation of remote method calls, care must also be taken, since the invoking object may be the functionality provider itself. To avoid the potential self deadlock and ensure that the process is performed in a safe way, a new actor is created at each remote invocation. This new object serves as a temporary one message queue for the source of the invocation and also becomes known by the target object. This helper actor not only handles a return value for the call but also stores the current message contents locally, waking up the originating object at the end of the process and resuming the original computation. In the beginning, the invocation message is dispatched to the helper actor, which packs the synchronous call in a way that is appropriate for target consumption and waits for a reply, which is dispatched back, when received, to the source object containing the original message arguments and the return value. The possible existence of a resuming point in the original computation is what reminds us to avoid the loss of computation context. We treat as follows the compilation of a method invocation and the respective reply, where $c = (\texttt{call}\ e_0\ MT^j[e_1, \ldots, e_m])$, $c \in b^k$, v^j and t^j are fresh identifiers z is a fresh constant:

```
behaviour Source
attributes a_1; ...; a_i;
MT^1 x_1^1 ... x_l^1 s^1 k^1 q^1: b^1;
...
---------------------------------------------
...
MT^j x_1^j ... x_m^j s^j k^j q^j: b^j;
...
---------------------------------------------
...
MT^k x_1^k ... x_n^k s^k k^k q^k: b^k;
...
```

$\pi \Downarrow (7)$

```
behaviour Source
attributes  a_1; ...; a_i; rb_j := 0; rv^j := nil;
MT^1 x_1^1 ... x_l^1 s^1 k^1 q^1: b^1;
...
---------------------------------------------
...
MT^j x_1^j ... x_m^j s^j k^j q^j: b^j;
...
---------------------------------------------
...
MT^k x_1^k ... x_n^k s^k k^k q^k: π(C(b^k, c));
  send (new MT^k_call[x_1^k, ..., x_n^k, s^k, k^k, q^k, self, z])
    MT^j[π(e_1), ..., π(e_m), π(e_0)];
_ x_1^k ... x_n^k s^k k^k q^k v^j t^j:
  if (t^j==z) then π(C̄(b^k, c)[c\v^j]);
```

This says that the synchronous call is substituted by a new actor creation, the dispatch of the request to this new actor and the treatment of the reply using a message tagged with _. We use above two functions C and $\overline{C}$ to compute respectively the customer behaviour up to the point before the remote invocation and its continuation. In our presentation, we have made some simplifying assumptions, including: that b^k is a list of commands without `if` branches and that there is just one synchronous call $c \in b^k$. The reader is refered to [Kim97] for a more detailed treatment. The following behaviour definition is included in our compiled code:

```
behaviour MT^k_call
attributes a_1; ...; a_n; s; k; q; o; z; d:= nil;
methods
MT^j x_1 ... x_m t: if (d==nil) then
  if (t==q) then begin
    d:=o; send o MT^j[x_1, ..., x_m, 1, self, self];
  end; else begin
    d:=t; send t MT^j[x_1, ..., x_m]@self;
  end; /* i.e. the customer of MT^j is self */
_ v : if ~(d==nil) then begin
    send d _[a_1, ..., a_n, s, k, q, z, v]; d:= nil;
  end;
```

The translations above are performed in a recursive manner and determine a set of derived behaviours from the given definitions. This assumes that there is only one method, necessarily with a disabling condition, treating each message type. This is not a serious restriction since message patterns can be abstracted, constants becoming variables in this case, unified and, together with their disabling conditions, treated. This is in effect part of our pre-compilation process.

The code generation step is interesting only when it comes to the translation of actor constructs, since constants, functions and their invocation, expressions and conditions can be directly rewritten in C. Each behaviour definition is translated into a procedure, where attributes are represented as local variables and the control flow determines how the actor reacts to incomming messages after its creation as a forked process. These independent control units make initial MPI calls to register themselves with the runtime environment. As a result, actors obtain unique identifications which serve as the extension of the addresses data type. Components are also registered as MPI objects so that they can dispatch initially present messages to the respective objects, although their identity is not available to programmers nor is any observable behaviour exhibited from this point onwards.

Concerning the translation of message passing, although MPI and LAM provide a multitude of modes including synchronous and asynchronous point-to-point interaction, the semantics of the actor model is most faithfully captured by the use of buffered local non-blocking sends and receives. These functionalities are requested as function calls, which are part of the generated code for the dispatch and delivery of messages. In the dispatch, the message contents are placed in a buffer (since all the manipulated data types are represented by integers, we are not even required to perform content packing). The sender may be allowed to continue its computation even before the end of this writing process. Our code generator guarantees that, just after its creation or the completion of a method body, each actor will be willing to accept new incoming messages. Whenever one arrives, the buffer containing the message tag and contents is used as a basis for the message pattern matching process.

The execution of a program presuposes the existence of a correctly configured fully functional cluster of computers based on LAM. There is no requirement on the type of hardware used, nor on the size of the cluster. The participant processing nodes are seen as a fully connected topology by the middleware. The MPI standard leaves process placement, migration and load balancing unspecified and Clara programs cannot rely on these facilities.

Programs may be executed in any node of a cluster. The first step in each execution is to check if the main component defined by the program can be composed with that containing all the already existing objects. This is an interactive process that requires the intervention of the program customer at first. The customer is asked to decide whether or not some of the new component externals can be bound to receptionists in the cluster and if some receptionists of the new component should be viewed as externals from the existing component perspective. In this way, a dynamic composition of the respective components is defined.

V. Concluding Remarks

In this paper we have presented Clara, a new programming language for high performance distributed computing. Clara is based on the actor model of distributed systems and inherits the conceptual clarity of this model due to the use of: (i) independent object-based units of control and computation, called actors; (ii) asynchronous point-to-point interaction, based on message passing; and (iii) (re)configurable units of coordination and modularisation, called distributed system components. The resulting language is powerful, yet simple. As a first step in our compilation process, we remove from each program synchronous constructs that are definable in terms of these three characteristic features of our kernel language (cf. [MT99]).

Many other languages and systems for high performance distributed computing have been proposed. ABCL [Yon90] and HAL [HA92] also had the actor model as a basis, but were proposed when it was not clear that, in order to define open distributed systems in terms of actors and obtain nice composability properties, a more structured notion of component is needed. Despite this, Clara is similar to HAL in that the two languages attempt to be independent of the operating environment by using publically available middleware, LAM MPI and Charm respectivelly. The concern with per-

formance issues was present not only in the design of our tool but also in the actor systems Rosette [TKS+89] and THAL [Kim97]. In addition, Clara shares with Concurrent Haskell [PJGF96] and Facile [GMP89] the asynchronous mode of interaction and the semantics clearly stratified into local computation and concurrency layers.

The comparison between Clara and the languages and systems above deserves further investigation in many respects, including their performance. We also believe that Clara is an appropriate basis for studying in the future program analyses and source level optimisations taking advantage of the simple core language and its well defined semantics.

Acknowledgements

The authors would like to thank Ian Mason for valuable suggestions for improving this paper. Part of this work was developed while Carlos was at the Department of Computing and Systems Engineering of IME, Instituto Militar de Engenharia, Rio de Janeiro. Carolyn was partially supported by the following research grants: NSF CCR-9900326, DARPA/NASA NAS2-98073 and ONR N00012-99-C-0198.

REFERENCES

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[BDV94] Gregory D. Burns, Raja B. Daoud, and James R. Vaigl. LAM: An open cluster environment for MPI. In *Proc. Supercomputing Symposium'94, 8th International Conference on High Performance Computing*, June 1994. http://www.mpi.nd.edu/lam/.

[DI96] Carlos H. C. Duarte and Roberto Ierusalimschy. On the systematic development of compilers: A case study. In Peter Fritzon, editor, *Proc. Poster Session of CC'96 - International Conference on Compiler Construction*, number LITH-IDA-R-96-12 in Technical Report Series of Department of Computer and Information Science, Linkoping University, pages 49–56, April 1996. Sweden.

[For97] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface.* July 1997. http://www.mpi-forum.org.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.

[HA92] Chris Houck and Gul Agha. HAL: A high-level actor language and its distributed implementation. In *Proc. 21th International Conference on Parallel Processing (ICPP'92)*, pages 158–165, August 1992.

[Kim97] Wooyoung Kim. *THAL: An Actor System for Efficient and Scalable Concurrent Computing.* PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[MT99] Ian Mason and Carolyn Talcott. Actor languages: Their syntax, semantics, translation and equivalence. *Theoretical Computer Science*, 228(1):277–318, 1999.

[PJGF96] Simon Peyton-Jones, Andrew Gordon, and Sigborn Finne. Concurrent Haskell. In *Proc. 23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308. ACM Press, January 1996.

[Tal98] Carolyn Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.

[TKS+89] Chris Tomlinson, Won Kim, Mark Scheevel, Vineet Singh, Becky Will, and Gul Agha. Rosette: An object oriented concurrent system architecture. *ACM SigPLAN Notices*, 24:91–93, April 1989.

[Tur86] David Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, December 1986.

[Yon90] Akinori Yonezawa. *ABCL: An Object-Oriented Concurrent System.* MIT Press, 1990.

APPENDIX

SIMPLIFIED ABSTRACT SYNTAX OF CLARA

$\mathbf{Var} \stackrel{\text{def}}{=}$ (variables)
$\mathbf{Attr} \stackrel{\text{def}}{=}$ (attributes)
$\mathbf{DefId} \stackrel{\text{def}}{=}$ (definition names)

$\mathbf{MsgTag} \stackrel{\text{def}}{=}$ (message tags) $(\_ \in \mathbf{MsgTag})$
$\mathbf{Id} \stackrel{\text{def}}{=} \mathbf{Var} \cup \mathbf{Attr}$ (identifiers)
$\mathbf{Const} \stackrel{\text{def}}{=} \mathbb{N} \cup \{\texttt{nil}\}$ (constants)

$\mathbf{Pat} \stackrel{\text{def}}{=} \mathbf{Id} \cup \mathbf{Const} \cup \{\texttt{self}, \texttt{cust}\}$ (patterns)
$\mathbf{ArithOp} \stackrel{\text{def}}{=} \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$ (arithmetic operators)
$\mathbf{BVar} \stackrel{\text{def}}{=} (\mathbf{Var}\ \texttt{:=}\ \mathbf{Expr})$ ("bound" variables)
$\mathbf{IAttr} \stackrel{\text{def}}{=} (\mathbf{Attr}\ [\texttt{:=}\ \mathbf{Const}])$ ([initialised] attributes)

Conditions and expressions:

$$\begin{aligned}\mathbf{Expr} \stackrel{\text{def}}{=}\ & \mathbf{Pat} \cup (\mathbf{Expr}\ \mathbf{ArithOp}\ \mathbf{Expr}) \cup \\ & (\mathbf{DefId}\ \mathbf{Expr}^*) \cup \\ & (\texttt{new}\ \mathbf{DefId}\ \mathbf{Expr}^*) \cup \\ & (\texttt{call}\ \mathbf{Expr}\ \mathbf{MsgTag}\ \mathbf{Expr}^*)\end{aligned}$$
$$\begin{aligned}\mathbf{Cond} \stackrel{\text{def}}{=}\ & (\mathbf{Expr}\ \texttt{==}\ \mathbf{Expr}) \cup \tilde{}\mathbf{Cond} \cup \\ & (\mathbf{Cond}\ \texttt{/\textbackslash}\ \mathbf{Cond}) \cup \\ & (\mathbf{Cond}\ \texttt{\textbackslash/}\ \mathbf{Cond})\end{aligned}$$

Commands:

$$\mathbf{Send} \stackrel{\text{def}}{=} \texttt{send}\ \mathbf{Expr}\ \mathbf{MsgTag}\ \mathbf{Expr}^*\ [\texttt{@}\mathbf{Expr}]$$
$$\begin{aligned}\mathbf{Cmd} \stackrel{\text{def}}{=}\ & \texttt{skip} \cup \mathbf{Send} \cup \\ & (\mathbf{Attr}\ \texttt{:=}\ \mathbf{Expr}) \cup \\ & (\texttt{reply}\ \mathbf{Expr}) \cup \\ & (\texttt{if}\ \mathbf{Cond}\ \texttt{then}\ \mathbf{Cmd}\ [\texttt{else}\ \mathbf{Cmd}]) \cup \\ & (\texttt{begin}\ \mathbf{Cmd}^*\ \texttt{end})\end{aligned}$$

Definitions:

$$\mathbf{GDef} \stackrel{\text{def}}{=} (\mathbf{Expr}\ [\mathbf{Cond} \cup \texttt{otherwise}])^+$$
$$\begin{aligned}\mathbf{MDef} \stackrel{\text{def}}{=}\ & (\mathbf{MsgTag} \cup \texttt{otherwise})\ \mathbf{Pat}^* \\ & [\texttt{disableby}\ \mathbf{Cond}]\ \mathbf{Cmd}^+\end{aligned}$$
$$\begin{aligned}\mathbf{BDef} \stackrel{\text{def}}{=}\ & \texttt{behaviour}\ \mathbf{DefId} \\ & [\texttt{attributes}\ \mathbf{IAttr}^*] \\ & [\texttt{methods}\ \mathbf{MDef}^*]\end{aligned}$$
$$\begin{aligned}\mathbf{CDef} \stackrel{\text{def}}{=}\ & \texttt{component}\ \mathbf{DefId} \\ & [\texttt{externals}\ \mathbf{Var}^*] \\ & [\texttt{receptionists}\ \mathbf{Var}^*] \\ & [\texttt{actors}\ \mathbf{BVar}^*] \\ & [\texttt{messages}\ \mathbf{Send}^*]\end{aligned}$$
$$\begin{aligned}\mathbf{FDef} \stackrel{\text{def}}{=}\ & \texttt{function} \\ & (\mathbf{DefId}\ (\mathbf{Var} \cup \mathbf{Const})^*\ \mathbf{GDef})^+\end{aligned}$$
$$\mathbf{Def} \stackrel{\text{def}}{=} \mathbf{FDef} \cup \mathbf{BDef} \cup \mathbf{CDef}$$
$$\mathbf{Unit} \stackrel{\text{def}}{=} \mathbf{Def}^* \cup (\texttt{program}\ \mathbf{DefId})$$
