Mobile Objects and Modern Communication Abstractions: Design Issues and Denotational Semantics

Eneia Nicolae Todoran, Claudiu Adam, Mirel Bâlc, Radu Pop,
Răzvan Radu, Dorin Simina, Emanuel Varga, Dan Andrei Zaharia
Department of Computer Science
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
Email: eneia.todoran@cs.utcluj.ro, adam.claudiu86@gmail.com,
mirel.balc@yahoo.com, pop.radu10@gmail.com,
razvansr@yahoo.com, dorin.simina@yahoo.com,
emanuelvarga@me.com, dan_andrei_zaharia@yahoo.com

***Abstract*—We introduce Join Voyager - a language that can be used to program a Peer to Peer network using object oriented techniques and Join methods. In Join Voyager any object can migrate to any node (peer) of the underlying network. The language provides strong mobility, i.e. the ability to capture and transfer the full execution state of mobile objects at any time. Objects can communicate by sending and receiving messages in object oriented style. As in Join Java or Polyphonic C#, both synchronous and asynchronous messages are supported. The paper presents the design rationales and a semantic interpreter for Join Voyager. The interpreter is designed with continuations following the discipline of denotational semantics and is implemented in Haskell.**

I. Introduction

The object oriented programming paradigm provides an appropriate framework for modeling interaction patterns and behaviors taken from real life scenarios. Real life objects can do various things. They can interact, but they can also move, i.e. they are *mobile*. Also, their mobility does not alter their interaction capability. For example, people can communicate by using mobile phones (and they can also do various other things in parallel) while they roam. Most object oriented programming languages do not provide direct support for modeling such complex behaviors, either because objects are not mobile or because each object is a sequential entity, unable to perform different tasks in parallel.

This paper presents Join Voyager - a language that can be used to program a Peer to Peer (P2P) network by using the modern communication abstractions introduced in Join Java [10] and Polyphonic C# [2]. In Join Voyager any object can migrate freely to any node of the underlying network and can communicate by sending and receiving messages in object oriented style. Each object is capable to answer in parallel an arbitrary number of messages. The interaction style is based on the Join method model introduced in Join Java and Polyphonic C#, which in turn are based on the Join calculus [5]. Both synchronous and asynchronous messages are supported. The communication capability of an object is not altered by its mobility or by the mobility of other objects. The language provides *strong mobility*, i.e. the ability to transfer the full execution state of mobile objects at any time rather than just at specific pre-determined points.

The semantic model of Join Voyager assumes the existence of a middleware for P2P computing that provides the following minimal set of services (1) a mechanism for naming the nodes (peers), (2) a feature for (asynchronous) transmission of messages between nodes and (3) a primitive that computes the list of nodes of the P2P network. For example, the JXTA platform [7] (developed at Sun Microsystems) provides the three features mentioned above. In the sequel we prefer the term *node*, rather than *peer*, but it is assumed that any two nodes can communicate in P2P style.

We present a denotational (compositional) semantics for Join Voyager. The semantic model is designed with continuation semantics for concurrency (CSC) [14], [15], monads and standard fixed point definitions. Instead of using a mathematical notation, we find convenient to use the lazy functional programming language Haskell [19] as a metalanguage for the denotational semantics. In this way, we allow our denotational model to be directly implementable, in the form of a semantic interpreter for the language under study, and thus to be easily tested and evaluated. At the same time, we avoid unnecessary complexities accompanying the use of domain theory [6] or the theory of metric spaces [1], which could have been adopted alternatively. The Join Voyager system comprises a static type checker and a semantic interpreter which implements a dynamic denotational semantics. In this paper we only present the semantic interpreter.

CSC is a general tool for designing denotational models of concurrency in interleaving semantics. The semantics of each statement is defined with respect to a *continuation*. The central characteristic of the CSC technique is the modeling of continuations as (application-specific) structures of computations (partially evaluated denotations). The structure of a CSC continuation is representative for the control concepts of the language under study.

The semantic prototype that we present in this paper is available from [20] in two variants: one that produces all possible execution traces of a program, and another one that produces incrementally a single execution trace. The first variant is *not* tractable (it can only be tested on toy programs). The second variant simulates the non-determinism of a "real" distributed program by using a (pseudo-)random number generator; this variant is *tractable* and reasonably efficient. It can be tested with "real life" programs. In this paper we present a distributed generator of prime numbers based on the sieve of Erathostenes. that uses mobile objects to print the result on all the nodes of the underlying P2P network. We have also tested successfully our semantic interpreter on all the algorithms based on Join methods given in [10] and we extended some of these algorithms with mobile objects [16].

1) Contribution: A significant number of languages have been created or simply adapted to incorporate some form of code mobility. For a comprehensive survey of mobile code languages, the reader may consult [4]. The key role played by the object oriented paradigm in the last decade is beyond dispute [3], [13]. The design of Join Voyager appears to be original. It combines the strong mobility of objects with the Join method model introduced in Join Java [10] and Polyphonic C# [2]. Also, the denotational model given in this paper is new. It uses various techniques, including continuations, monads and fixed point definitions. To the best of our knowledge this is the first paper that presents a compositional semantics for a strong mobility feature at object level combined with the modern communication abstractions introduced in Join Java and Polyphonic C#. Due to space limitations the paper only presents the semantic domains and a selection of the semantic equations. A complete version of the paper is available online as a technical report [16].

II. The language Join Voyager

A. Main design issues

It is generally considered that the P2P model represents a promising paradigm for global computing, offering a scalable pragramable infrastructure [18]. However, the design of programming languages that exploit the flexibility provided by the P2P model is still at the beginning [9]. The solution that we propose in this paper is developed within the framework of the object oriented paradigm. It is based on the idea of combining a strong mobility feature at object level with the modern communication abstractions introduced in Join Java [10] and Polyphonic C# [2]. Our solution also takes into consideration general software engineering principles for achieving a good design. In particular, we strive to achieve a model that encourages module cohesion.

As far as we know, the only (alternative) existing solution to exploiting the flexibility provided by the Join method model in a distributed setting is based on the *movable* methods of MC# [8]. The problem with that solution is that in object oriented programming a method is *not* a unit of protection. Our solution obeys a a general software engineering principle, commonly known as *communication cohesion* [11]. A system exhibits communication cohesion when all modules that access or manipulate certain data are kept together (e.g. in the same class), and everything else is kept out. One of the strong points of the object-oriented paradigm is that it helps ensure communicational cohesion.

The language Join Voyager provides a primitive $\mathsf{go}(e)$ which can be used to initiate the migration of the current object (the migration of self). e is an expression that must evaluate to a valid node name π. The object that executes this statement will move physically to node π. Such a migration can be initiated from outside (by any another object) through any method of the current object.[1] In the Join Voyager model the distribution unit is the object rather than the method. Remote references toward a mobile object remain valid no matter where the object migrates.

Join Voyager embodies both the mobility of objects and the mobility of object references. Any message can carry parameters that are object references (rather than the objects themselves). In addition, any object can migrate on any node and thus any service can be executed on any node. The callers of different Join method fragments can be located on different P2P nodes. For a synchronous Join method the result is returned to the caller of the synchronous fragment.

Another important decision in the design of Join Voyager is based on the idea that an object should be able to create an arbitrary number of threads to serve incoming messages. Each Join Voyager thread is created in response to a (local or remote) method invocation. The threads created by an object share the instance variables of the object, but each thread also has its own temporary variables that are *not* shared with other threads. Any Join Voyager object can answer in parallel an arbitrary number of messages. Mutual exclusion can be programmed as in Join Java or Polyphonic C#.

B. Syntax and informal explanation

A Join Voyager *object* is an instance of a *class*. Each object has a (unique) *name*. For an object that is an instance of an user defined class its name can be used as an *identifier* or a *reference* to the object. A message can carry one or several object names as parameters. A class definition is a collection of synchronous and asynchronous Join methods. A Join method is composed of Join method fragments that can be synchronous or asynchronous. A synchronous Join fragment can return to the caller an answer that is an object name. In the sequel we assume given a set $(i \in)Ivar$ of *instance variables*, a set $(v \in)Tvar$ of *temporary variables*, a set $(C \in)Cname$ of *class names*, a set $(f \in)Jfname$

[1]For simplicity, in this semantic study all the methods are *public*.

of *Join fragment names* and a set $(\pi \in) Nname$ of *node names* (or node identifiers).[2] One defines a set $(\phi \in) Sobj$ of *standard objects* as follows:

$$(\phi \in) Sobj = \mathbb{Z} \cup Bool \cup Nname \cup Nname^* \cup \{nil\}$$

A standard object can be an integer number $(z \in)\mathbb{Z}$, a boolean value $(b \in) Bool = \{\,\mathsf{true}\,,\,\mathsf{false}\,\}$, a node name, a list of node names, or the special object nil. We put $Nname = \mathbb{N}$. The name of a node is a unique identifier (different nodes have different names). It is assumed that the underlying network includes a node with name (identifier) 0; any Join Voyager program starts its execution on this node.

Standard objects answer to messages according to the intuitive semantics. Any method of a primitive object is synchronous and is composed of a single fragment. The objects of type $(\varpi \in) Nname^*$ are lists of node names and deserve a special attention. For an object $\varpi(\in Nname^*)$ the programmer can use the following methods. In each case, the type T of the value returned by a method call $\varpi.m()$ is indicated by using the notation: $\varpi.m() : T$.

$$\varpi.readNet() : Nname^*, \quad \varpi.isEmpty() : Bool$$
$$\varpi.first() : Nname, \quad \varpi.rest() : Nname^*$$

A call $\varpi.readNet()$ returns the list of all nodes allocated to the program. In the sequel we assume that this primitive is deterministic. In our implementation a call $\varpi.readNet()$ returns an ordered list (with respect to the natural order over $\mathbb{N}$). In case the network is composed of 5 nodes with names $0, 1, 2, 3$ and 4, a call $\varpi.readNet()$ will always return the list $[0, 1, 2, 3, 4]$. For simplicity, we ignore issues such as node creation, destruction or failure. A call $\varpi.isEmpty()$ returns true if ϖ is the empty list, and false otherwise. The empty list $(\in Nname^*)$ is denoted by the symbol ϵ. An expression $\varpi.first()$ reduces to the name of the first node of the list ϖ. An expression $\varpi.rest()$ reduces to the list resulted by removing the first element of the list ϖ.

The syntax of Join Voyager is given below in BNF. A program $p(\in Prg)$ is a list of class declarations. A class declaration is a pair consisting of a class name C and a class definition c. A class definition is a triple, consisting of a list of synchronous method definitions, a list of asynchronous method definitions, and a statement whose execution is started automatically upon the creation of a class instance (a new object). The execution of a Join Voyager program is started by the creation of an instance of the last class defined in the program on the node with name (identifier) 0.

As in Join Java [10], the definition of an asynchronous method is preceded by the keyword signal . A (synchronous or asynchronous) Join method definition consists of a header, that is a conjunction of Join fragments $(j \in Jh)$ and the body of the method. The body of a synchronous method is an expression; the value of the expression is returned by the Join method to the caller of the synchronous fragment.

[2]In this paper the notation $(x, y, \cdots \in) X$ introduces the set X with typical variables $x, y, \cdots$.

As in Join Java [10] only the first (leftmost) fragment can be synchronous. The body of an asynchronous method is a statement; in this case nothing is returned to the caller.

$$\begin{array}{rl}
prg(\in Prg) ::= & \mathsf{class}\ C\ \chi \cdots \mathsf{class}\ C\ \chi \\
\chi(\in Cdef) ::= & \{\, \rho \cdots \rho \ [\!]\ \varrho \cdots \varrho \ [\!]\ s \,\} \\
\rho(\in Jsdef) ::= & j\,\{\,e\,\} \\
\varrho(\in Jadef) ::= & \mathsf{signal}\ \ j\,\{\,s\,\} \\
j(\in Jh) ::= & f(v, \cdots, v) \mid j\ \ \&\ j \\
e(\in Exp) ::= & \mathsf{self}\ \mid \phi \mid i \mid v \mid \mathsf{new}(C) \mid e = e \\
 & \mid s; e \mid e\,.\,f(e, \cdots, e) \\
s(\in Stat) ::= & \mathsf{skip}\ \mid i := e \mid v := e \mid \mathsf{exp}(e) \mid \mathsf{write}(e) \\
 & \mid \mathsf{go}(e)\ \mid s; s \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s \\
 & \mid \mathsf{while}\ e\ \mathsf{do}\ s\ \mid e\#f(e, \cdots, e) \mid \{s\}
\end{array}$$

An *expression* $e(\in Exp)$ is a syntactic construct that reduces to a value, i.e. an object name. An *object name* is either a standard object or a reference to an object which is an instance of a user defined class. The expression self reduces to the name of the object for which it is evaluated. The evaluation of a standard object produces the object itself. The semantics of an instance or a temporary variable is the value (object name) stored in it. An expression $(e_1 = e_2)$ reduces to a boolean value: true if e_1 and e_2 reduce to the same object name, respectively false, otherwise. An expression $s; e$ evaluates the statement s for its side effects; the value of $s; e$ is given by the value of e. The expression $\mathsf{new}(C)$ creates a new instance of class C (on the local node), and produces as value a reference to the new object.

An expression $e\,.\,f(e_1, \cdots, e_n)$ is a synchronous Join fragment call; it has blocking semantics. The expression e must reduce to a (local or remote) object name which is the destination of the message. The expressions $e_1, \cdots, e_n$ must also reduce to object names which are the parameters of the call. Only the first (leftmost) fragment of the header of a synchronous Join method is synchronous. The body of a synchronous Join method is an expression, which is evaluated only when all Join fragment calls arrive. The result is returned to the caller of the synchronous Join fragment.

A Join Voyager statement is a syntactic construct that is only evaluated for its side effects (it produces no value). skip is the inoperative statement. $i := e$ and $v := e$ are assignment statements. An expression e can be treated as a statement $\mathsf{exp}(e)$, in which case its value is ignored. The statement $\mathsf{write}(e)$ prints the value of the expression e at the standard output device of the node on which the object that executes the $\mathsf{write}(e)$ statement is physically located. Join Voyager statements can be combined by using constructs for sequential composition $(s; s)$, conditional (if) and repetitive (while) execution. $e\#f(e, \cdots, e)$ is an asynchronous Join fragment call that does not block the caller. Such a call returns immediately. The body of an asynchronous Join method is executed in a separate thread.

The primitive $\mathsf{go}(e)$ can be used to initiate the migration of the current object (the migration of self). e is an expression that must evaluate to a valid node name π. This operation captures the complete execution state of the current object (its instance variables, the collection of Join fragment calls and the collection of threads that serve all its current Join method invocations) at the moment when the migration operation is initiated. An object that executes this statement moves physically to the destination node π. Its execution is resumed at the destination node in the state it was when the migration operation was initiated.

1) A toy program: The execution of the Join Voyager program given below starts on node 0 by the (automatic) creation of an object of type Toy. First, this object prints the value 1 on node 0. Next, it creates a thread by an asynchronous Join method call $\mathsf{self}\,\#f(2)$. This thread is executed in parallel with the rest of the program $\mathsf{go}(2)$; $\mathsf{write}(3)$. In case the thread is faster than the rest of the program it will print the value 2 on node 0. Otherwise, a migration operation is initiated by the statement $\mathsf{go}(2)$ and the thread will migrate together with the rest of the object. The execution of the object is resumed on node 2. We verified this program by using our interpreter in "all possible traces" semantics and we obtained the following set of (possible) traces: $[[(1,0),(3,2),(2,2)],[(1,0),(2,0),(3,2)],[(1,0),(2,2),(3,2)]]$. In our implementation a trace is a list of pairs of the form (observable value, node name). For example, the trace $[(1,0),(3,2),(2,2)]$ corresponds to the following execution scenario. After printing the value 1 on node 0, the instance of Toy migrates on node 2; on node 2 the execution of statement $\mathsf{write}(3)$ is faster than the thread that was created by the asynchronous call $\mathsf{self}\,\#f(2)$.

```
class $Toy${
[] signal $f(v)${ write($v$) } [] write(1) ; self $\#f(2)$; go(2) ; write(3)
}
```

We also tested the above program by using the "single trace" variant of our semantic interpreter. The (pseudo-)random number generator that is used to model the non-determinism in "single trace" semantics is initialized with different seeds at consecutive executions. Therefore, in general, a non-deterministic program produces different execution traces at consecutive executions. For the above (toy) program all the three possible traces are produced by our interpreter after the first eight consecutive executions in "single trace" semantics: $[(1,0),(2,2),(3,2)]$, $[(1,0),(2,0),(3,2)]$, $[(1,0),(2,0),(3,2)]$, $[(1,0),(2,0),(3,2)]$, $[(1,0),(2,0),(3,2)]$, $[(1,0),(2,0),(3,2)]$, $[(1,0),(2,0),(3,2)]$, $[(1,0),(3,2),(2,2)]$.

2) Sieve of Erathostenes: We present a distributed generator of prime numbers based on the sieve of Erathostenes. The prime numbers are generated on a single node (on the initial node 0) and are printed on all the nodes of the network. A $Sieve$ object is created for each prime number. Each $Sieve$ object creates a $Printer$ object which migrates and prints the corresponding prime number. Each $Printer$ object is created in the body of the asynchronous Join method $\mathsf{signal}\ print()\ \&\ loc(v_{net})$, therefore performance is *not* altered by the distributed printing process. The observable behavior of this program is deterministic. The generator produces the stream of prime numbers in strict increasing order and each $Printer$ object traverses the nodes of the network in the same order (given by the initial call to $readNet()$). Also, each $Printer$ object completes its printing job before the next $Printer$ object is created.

$Sieve$ objects are synchronized by using the Join fragment calls to $input()$ and $answerInput()$ and the auxiliary class RV, following the rendezvous pattern given in section 3.2 of [2]. For each call to $input()$ an instance of RV is created, in order to $wait$ for an asynchronous $reply$ message, which is sent after synchronization with $answerInput()$.

```
class $RV${
  $wait()\ \&\ reply()\{self\}$ [] [] self
}
class $Sieve${
  $input(v_n)\{$
    $v_{rv} := new(RV)$; self $\#ainput(v_n, v_{rv})$; $v_{rv}\,.\,wait()$
  }
  $answerInput()\ \&\ ainput(v_n, v_{rv})\{$
    $v_{rv}\#reply(); i_q := v_n$; self
  }
[] signal $print()\ \&\ loc(v_{net})\{$
    $v_{prn} := new(Printer)$; exp($v_{prn}\,.\,print(i_p, v_{net})$);
    $i_{next}\#loc(v_{net})$
  }
[] exp( self . $answerInput()$); $i_p := i_q$;
  $i_{next} := new(Sieve)$; self $\#print()$;
   while ( true ){
     exp( self . $answerInput()$);
     if $(((i_q.mod(i_p)).eq(0)).not())$ then exp($i_{next}\,.\,input(i_q)$)
                                              else skip ;
   }
}
class $Printer$
  $print(v_p, v_{net})$ {
     if ($v_{net}\,.\,isEmpty()$) then skip
     else { go($v_{net}\,.\,first()$) ; write($v_p$) ;
              exp( self . $print(v_p, v_{net}\,.\,rest())$)
     }; self
  }
[] [] skip
}
class $Primes${
[] [] $i_{first}$ := new($Sieve$) ; $v_{net} := \epsilon\,.\,readNet()$;
     $i_{first}\#loc(v_{net})$; $i := 2$;
     while ($i\,.\,leq(20)$){ exp($i_{first}\,.\,input(i)$); $i := i\,.\,add(1)$ }
}
```

This program generates all prime numbers less than 20. It cannot be verified in "all possible traces" semantics. We tested this program by 1000 consecutive executions of the "single trace" variant of our interpreter. A network with 5 nodes $(0, 1, 2, 3$ and $4)$ was considered for testing purposes. The semantic interpreter always produced the following execution trace: $[(2,0),(2,1),(2,2),(2,3),(2,4),(3,0),(3,1),(3,2),(3,3),(3,4),\hookleftarrow$ $(5,0),(5,1),(5,2),(5,3),(5,4),(7,0),(7,1),(7,2),(7,3),(7,4),\hookleftarrow$ $(11,0),(11,1),(11,2),(11,3),(11,4),(13,0),(13,1),(13,2),\hookleftarrow$ $(13,3),(13,4),(17,0),(17,1),(17,2),(17,3),(17,4),(19,0),\hookleftarrow$ $(19,1),(19,2),(19,3),(19,4),deadlock]$. In this experiment the same trace was obtained by 1000 consecutive executions in "single trace" semantics (at each execution the random number generator is initialized with a different seed). This can give a good degree of confidence that the program is correct and its observable behavior is deterministic.

III. Semantic prototyping with continuations for concurrency and monads

In recent work [15] we showed that, by using CSC continuations and monads denotational semantics can be used both as a method for formal specification and as a general method for designing compositional prototypes of parallel and distributed languages. Monads, which are directly supported in Haskell [19], are well known as a tool for improving the modularity and elegance in denotational descriptions and functional programs [17]. The CSC technique was introduced in [14], [15]. It provides *flexibility* in the denotational design of concurrent control concepts [15]. The central characteristic of the CSC technique is the modeling of continuations as structured configurations of computations, where by *computation* we understand a partially evaluated denotation (meaning function).

In the CSC approach the space of computations is divided into one *active* computation and the rest of the computations which are encapsulated in a *continuation*. Intuitively, the CSC technique is a semantic formalization of a process scheduler [14]. Each computation remains active only until it performs an elementary action; subsequently, another computation taken from the continuation is planned for execution. In this way it can be obtained the desired interleaving behavior for parallel composition.

CSC continuations can be divided into two categories: *closed continuations* and *open continuations*. An open continuation is a structure of computations which contains a *hole* (indicating the conceptual position of the active computation). An open continuation behaves as an *evaluation context* for the active computation. A closed continuation is a self contained structure of computations. CSC continuations are only closed for scheduling purposes. In [15] we designed various CSC continuation structures as combinations of two basic concepts: the *stack* to model sequential composition and the *multiset* (a collection in which an element may occur more than once) to model parallel composition.

The functions of an CSC-based semantic interpreter can be grouped into the following three components: an *evaluator*, a continuation-completion mapping with a *normalization procedure*, and a *scheduler*. An CSC-based interpreter implements an "evaluate-normalize-schedule" loop [15].

- The evaluator maps open continuations to program behaviors. It comprises the (compositional) definition of the denotational function together with language-specific control operators. *The functions of the evaluator have one thing in common: they manipulate open continuations (i.e. evaluation contexts) only*.
- The continuation-completion function is called by the evaluator to map an open continuation to the program answer that would result if the continuation alone were left to execute. First, it calls *the normalization procedure, which transforms the open continuation into a corresponding closed continuation* (intuitively, by removing the 'hole' from the open continuation); next it calls the scheduler.
- *The scheduler maps closed continuations to program behaviors*. It activates a computation by decomposing a closed continuation into an (activable) computation and a corresponding open continuation. For a distributed language the selection of the activable computation may be nondeterministic, and it may follow after a (finite) number of internal synchronization steps.

In the CSC approach the final yield of the denotational semantics can be encapsulated in a program behavior monad, which can be designed in "single trace" or "all possible traces" semantics. To obtain a "single trace" model, the program behavior monad is parameterized by a random number generator which is used to simulate the non-determinism of a "real" distributed language. In this case, the semantic interpreter is *tractable* and produces incrementally a single execution trace. The "all possible traces" model is obtained by using a classic power domain monad in this case the resulted interpreter is generally *not* tractable. An element of a power domain [12] is exponential to the length of execution traces (an element of a power domain is a tree-like structure, or a collection of "traces" essentially equivalent to an unfolding of such a tree).

IV. Semantic interpreter for Join Voyager

A. Semantic domains

A denotational semantics is a compositional mapping that associates values from a mathematical domain [6], [1] to each program construct. In order to obtain an executable specification instead of using a mathematical notation, we find convenient to use Haskell [19] as a metalanguage for the denotational semantics.

1) Object names, states and observables: The Haskell definitions for object names, states and observables are given in figure 1. `Nname` implements the set of node names. `Ivar`, `Tvar` and `Cname` implement the sets of instance variables, temporary variables and class names, respectively. `Nname` is a type synonym for `Int` (`type Nname = Int`). `Ivar`, `Tvar` and `Cname` are type synonyms for `String`.

The type `ObjN` implements the set of *object names*. An object name is either a standard object (of type `Sobj`) or a reference to an instance of a user defined class. An `Oref` object reference is a pair consisting of a class name and an `Id` *identifier* which is *unique* at distributed system level. An `Id` identifier is a pair `(i,nn)` consisting of a number `i` (`i` is unique at node level) and a node name `nn`. `nn` is the name of the node where the object was created, its *parent node*. The parent node maintains references to all objects that are created on it, including references to remote objects.

Figure 1. Object names, states and observables

```
type Id = (Int,Nname)
type Oref = (Cname,Id)
data Sobj = Nil | B Bool | Z Int | N Nname | Net [Nname]
data ObjN = Oref Oref | Sobj Sobj
type S1 = Ivar -> ObjN
type S2 = Tvar -> ObjN
data Q = Epsilon | Deadlock | Q Obs Q
data Obs = ObsZ Int Nname | ObsB Bool Nname | ...
```

The communication protocol uses this information to locate destination objects [16]. A value of type `S1` (`S2`) is a store of instance (temporary) variables. The type `Q` implements a domain of sequences of observables and distinguishes between normal termination and deadlock.[3]

2) Denotations and continuations: The domain of denotations is defined in figure 2. To avoid notational overhead by allowing an arbitrary number of arguments for the constructs $e\,.\,f(e,\cdots,e)$ and $e\#f(e,\cdots,e)$, in section IV we restrict the model to messages with exactly one parameter, of the form $e\,.\,f(e)$ and $e\#f(e)$, respectively.

`D` is the domain of *denotations*. `M` is the program behavior monad described in subsection IV-B. `Cont` is the domain of *open continuations*. `Kont` is the domain of *closed continuations*. Both open and closed continuations are sets of nodes; for an open continuation we separate a particular node (the local node) to make more obvious that the evaluator only acts upon this node. Only the scheduler (described in IV-E) manipulates closed continuations and handles communications between different nodes. `F` is the domain of *expression continuations*, which facilitate the evaluation of expressions in classic continuation-passing style.

A node `(Node nn po pd pr obuf mbuf)` has a name `nn`, a set `po` of active objects, a set `pd` of dormant objects, a set `pr` of references to remote objects, and two multisets of packages `obuf` and `mbuf` handled by the mobility and communication protocol, respectively. We implement sets, multisets and stacks as Haskell lists.

An *active object* `(Aobj n pt pj w s1)`, has a name `n`, a multiset `pt` of threads, a multiset `pj` of incoming messages (of type `Msg`), a rendezvous counter `w` (of type `Wait`) and a store of instance variables `s1`. By using the rendezvous counter and by capturing the thread continuation of a synchronous Join fragment call in its expression continuation we avoid introducing thread identifiers. A thread is a pair consisting of a list (implementing a stack) of computations and a store of temporary variables. In our implementation, the 'hole' (that indicates the conceptual position of the active denotation) is at the head (the 'leftmost' element) of the list that implements a stack, a set or a multiset. For an open continuation `(node,k)` the 'hole' is at the head of the leftmost thread of the leftmost active object of node `node`. It is the task of the scheduler to produce such a configuration in preparation for the evaluation of a computation.

When the components `pt` and `pj` of an active object become empty lists and the rendezvous counter becomes `NoWait` the object becomes dormant. A *dormant object* `(Dobj n s1)` is characterized by its name (reference) `n` and its instance variables store `s1`. If such a dormant object receives messages it becomes active again.

Each (parent) node also maintains a list `pr` of references to all remote objects that were created on it but are currently somewhere else. Such references are elements of type `Robj`. We use a structure `(Robj n nn)` to express that the object with name `n` is currently located on the node with name `nn`. A structure `(OnMove n)` expresses that the object with name `n` is currently involved in a migration operation.

A computation of type `Comp` is either a denotation `(D d)`, or an attempt `(SemSend n semcall)` to transmit a semantic call `semcall` to object `n`, or an attempt `(SemGo nn)` of the current object to start a migration operation toward node `nn`. A `SemCall` semantic call may be synchronous or asynchronous; it always carries information of type `Match` and `Env`. Values of type `Match` are used in the implementation of the pattern matching mechanism that is specific of languages based on the Join calculus. The type `Jfname` implements the set of Join fragment names. A value of type `Env` is a *semantic environment* which captures the semantics of Join methods (see section IV-F). In the case of a synchronous method the expression continuation of the synchronous Join fragment call is also transmitted with the semantic call.

The communication and mobility protocols use packages of type `MsgVoyager` and `ObjVoyager`, respectively, to transmit information between nodes. A package `(MsgVoyager n msg)` contains a reference `n` to the destination object and a synchronous or asynchronous message `msg :: Msg`. A value of type `Msg` carries information that is specific of a semantic call; in addition, in case of a synchronous call, it also carries a reference `n'` (back) to the object which initiated the synchronous (Join fragment) call and it captures the thread continuation `t` of the synchronous call. Thus, the thread that initiated the synchronous call is suspended during the evaluation of the corresponding (synchronous Join) method. Upon termination of execution of the corresponding method the thread continuation `t` is released and transmitted back to the object `n'` which initiated the call by using a package `(ReleaseVoyager n' t)`. The rendez vous counter (`w`) of the object which initiates a synchronous call is incremented (`Wait w`) when the call is submitted and it is decremented (`w`) when the execution of the corresponding method completes.

The mobility protocol uses a package of the form `(ObjVoyager nn' mbuf obj)` to move an object `obj` to a destination node `nn'`. The package carries all the messages[4]

[3]There is no universal algorithm for distributed termination or deadlock detection. In [16] it is explained how to add a halt statement to Join Voyager which stops a complete distributed program by using a very simple protocol. Otherwise the program runs indefinitely on all nodes.

[4]Join fragment calls which wait for other Join fragment calls to arrive in order to complete a method call by a successful Join pattern matching.

Figure 2. Denotations and continuations

```
type D = Cont -> M Q
type Cont = (Node,[Node])
type Kont = [Node]
type F = ObjN -> D

data Node = Node Nname [Aobj] [Dobj] [Robj] Obuf Mbuf
data Aobj = Aobj Oref [Thread] [Msg] Wait S1
data Dobj = Dobj Oref S1
data Robj = Robj Oref Nname | OnMove Oref
type Obuf = [ObjVoyager]
type Mbuf = [MsgVoyager]
type Thread = ([Comp],S2)
data Wait = NoWait | Wait Wait
data Comp = D D | SemSend Oref SemCall | SemGo Nname
data SemCall = SemSynCall Match Env F
             | SemAsynCall Match Env
type Match = (Jfname,ObjN)

data MsgVoyager = MsgVoyager Oref Msg
                | ReleaseVoyager Oref Thread
data Msg = Msg Match Env SynInfo
data SynInfo = SynMsg F Oref Thread | AsynMsg
data ObjVoyager = ObjVoyager Nname Mbuf Obj
                | ProtocolVoyager ProtocolInfo
data ProtocolInfo = Request Oref Nname | Accept Oref Nname
                  | Completion Oref Nname
data Obj = ObjA Aobj | ObjD Dobj
```

that have not been served yet by `obj` in a list `mbuf :: Mbuf`. An object may travel as a dormant object in case after the go(e) statement it has no other statement to execute until it receives other messages. A migration operation from a source node `nn` to a destination node `nn'` is performed by an object with name `n` with the permission of its *parent* node. The object sends a package `(Request n nn)` to its parent node.[5] Upon receiving this package the parent node records the status `(OnMove n)` for the object `n`; next it sends a package `(Accept n nn)` back to object `n` on node `nn`. The object then migrates to the destination node `nn'`; when it arrives at the destination it sends a package `(Completion n nn')` to its parent node. The migration operation is completed by the parent node which changes the status of object `n` from `(OnMove n)` to `(Robj n nn')`.

B. Program behavior monad

Our semantic interpreter is parameterized by a program behavior monad `M` which is designed in two ways, roughly corresponding to the two perspectives on nondeterminism described in the last paragraph of section III. The denotational semantics is of the type `type D = Cont -> M Q`, where `Cont` is the domain of (open) continuations and `Q` is a a domain of execution traces (sequences of observables). The monad `M` is used to encapsulate the final yield (`M Q`) of the denotational semantics. It is remarkable that only this final yield of the denotational mapping distinguishes between the two ways of interpreting the nondeterminism.

The monad for "all possible traces" semantics implements a continuation semantics using powerdomains (a behavior is a collection of traces and a nondeterministic choice is a union of behaviors). This monad allows one to detect whether a particular execution trace is possible, but it can only be used to verify toy programs. To implement the "single trace" semantics, the monad is parameterized by a random number generator (an *oracle*) that decides the alternative to be selected in nondeterministic choices (being given different oracles, *any* possible trace can be obtained). The "single trace" variant of our interpreter is *tractable* and can be used to test nontrivial distributed programs.

C. Evaluator

The evaluator is designed as two mutually recursive denotational functions: `seme :: Exp -> Env -> F -> D` and `sems :: Stmt -> Env -> D`. Its design is based on a combination of CSC continuations and classic continuation-passing style. The constructs go(e), $e\,.\,f_j(e')$ and $e\#f_j(e')$ are handled as follows. After the expressions involved are evaluated a corresponding computation is created which is added to the continuation in order to be processed by the scheduler. Assuming `(SynCall e j e')` is the Haskell implementation of the construct $e\,.\,f_j(e')$, the code given below defines the semantics of a synchronous Join fragment call. Standard objects are handled according to the intuitive semantics. For user defined objects the effect of such a call is given by the creation of a structure `(SemSend n ...)`, where `n` is a reference to the (local or remote) destination object. This structure is added to the open continuation `c` in the appropriate position (indicated by the conceptual 'hole', which is at the head of the list that implements the active thread) in order to be processed by the scheduler.

```
seme (SynCall e j e') env f c =
   seme e env (\o -> seme e' env (\o' -> aux o j o')) c
   where
      aux :: ObjN -> Jfname -> ObjN -> D
      aux (Sobj (Z z1)) "add" (Sobj (Z z2)) =
         f (Sobj (Z (z1 + z2)))
      ...
      aux (Oref n) j o'                     =
         let semcall = SemSynCall (j,o') env f
         in \c -> cc (addc (SemSend n semcall) c)
```

D. Continuation completion and normalization procedure

The completion function `cc` receives as input an open continuation `c` and calls the normalization procedure `re :: Cont -> Kont` in order to obtain the corresponding closed continuation `k` which is transmitted to the scheduler mapping `kc`. The semantic interpreter maintains the following invariant for open continuations: every thread is nonempty, with the possible exception of the leftmost one - the *active* thread - which conceptually contains at its head the active computation. The normalization procedure removes the active thread in case it becomes empty.

```
cc :: Cont -> M Q
cc c = let k = re c in kc k
```

E. Scheduler

The scheduler function `kc :: Kont -> M Q` maps closed continuations to program behaviors. The main components

[5]The system knows the destination of this package from the name of the object (`n`), which includes the name of its parent node.

Figure 3. Semantic environments

```
type Env = Cname -> (U1,U2,U3)
type U1 = D
type U2 = ([[Jfname]],G)
type U3 = ([[Jfname]],H)
type G = [Jfname] -> [ObjN] -> F -> D
type H = [Jfname] -> [ObjN] -> D
```

of the scheduler are two functions `cProtocol` and `mProtocol` which implement the steps of the communication protocol and mobility protocol, respectively. Also, an important component of the communication protocol is a function named `match` that implements the pattern matching semantics that is characteristic for languages based on the Join calculus [5].

The scheduler provides an accurate description of a P2P distributed implementation. The scheduler uses a single primitive for asynchronous transmission of protocol information between different nodes. All the other operations performed by the semantic interpreter are local to a particular node. Any synchronization effect (in particular the execution of a synchronous Join method) is obtained as a consequence of the distributed protocol steps.

F. Semantic environment and fixed point semantics

The domain of semantic environments `Env` is given in figure 3. A semantic environment is a mapping from class names to triples `(u1,u2,u3) :: (U1,U2,U3)`. For a given class the component `u1 :: U1` defines the semantics of the body, the component `u2 :: U2` defines the semantics of the synchronous Join methods, and the component `u3 :: U3` defines the semantics of the asynchronous Join methods.

Join method declarations may be recursive. The semantic environment `env :: Env` is defined as fixed point of a higher order mapping given in [16]. Haskell is a lazy functional language [19]. The fixed point operator can be defined as follows: `fix :: (a -> a) -> a, fix f = f (fix f).`

V. Concluding remarks and future research

We presented the main design issues and a dynamic denotational semantics for Join Voyager. Join Voyager is an experimental object oriented language that combines a strong mobility feature at object level with communication abstractions based on the Join method model [10], [2].

We introduced Join Voyager as a language for P2P programming. A distributed implementation of Join Voyager on top of the JXTA platform [7] is in progress. A limitation to P2P computing may not be mandatory, but we leave the adaptation of the Join Voyager programming model to other types of networks as a topic for future research.

Finally, fundamental research related to the CSC technique will be directed toward a complete formalization of the semantic interpreter given in this paper in the form of a corresponding (mathematical) denotational semantics.

References

[1] J.W. de Bakker, E.P. de Vink, *Control Flow Semantics,* MIT Press, 1996.

[2] N. Benton, L. Cardelli, C. Fournet, ”Modern concurrency abstractions for C#,” ACM Transactions on Programming Languages and Systems (TOPLAS), 25(5):769-804, 2004.

[3] D. Caromel, L Henrio, *A theory of distributed objects: asynchrony, mobility, groups, components.* Springer, 2005

[4] G. Cugola, C. Ghezzi, G. Picco and G. Vigna, ”Analyzing mobile code languages,” *LNCS*, 1222:93-111, 1997.

[5] C. Fournet, G.Gonthier, ”The Join calculus: a language for distributed mobile programming,” LNCS, 25:268- 332, 2002.

[6] G. Gierz et al, *Continuous lattices and domains.* Cambridge University Press, 2003.

[7] J.D. Gradecki, *Mastering JXTA: Building Java Peer-to-Peer applications,* Wiley, 2002.

[8] V. Guzev and Y. Serdyuk, ”Asynchronous parallel programming language based on the Microsoft .NET platform,” LNCS 2763:236-243, 2003.

[9] Q. Hieu Vu, M. Lupu and B. Chin Ooi, *Peer-to-Peer computing: principles and applications,* Springer, 2009.

[10] G.S. von Itzstein, Introduction of high level concurrency semantics in object-oriented languages, Ph.D.Thesis, University of South Australia, 2005.

[11] T. Lethbridge and R. Laganiere, *Object oriented software engineering (2nd edition),* Prentice-Hall, 2005.

[12] G.D. Plotkin, ”A Powerdomain Construction,” *SIAM Journal of Computing*, 5(3):452–487, 1976.

[13] R. Quitadamo, The issue of strong mobility: an innovative approach based on the IBM Jikes Research Virtual Machine, Ph.D. Thesis, University of Modena and Reggio Emilia, 2008.

[14] E.N. Todoran, ”Metric semantics for synchronous and asynchronous communication: a continuation-based approach,” Electronic Notes in Theoretical Computer Science, vol.28, pp. 119–146, Elsevier, 2000.

[15] E.N. Todoran and N. Papaspyrou, Continuations for prototyping concurrent languages, Technical Report CSD-SWTR-1-06, National Technical University of Athens, Software Engineering Laboratory, 2006.

[16] E.N. Todoran et al. Mobile objects and modern communication abstractions, Technical Report CSD-SE-TR-01-2011, Technical University Cluj-Napoca, 2011, available from [20].

[17] P. Wadler, ”The Essence of Functional Programming,” In *Proc. of 19th ACM Symposium on Principles of Programming Languages*, pages 1–14, ACM Press, 1992.

[18] K. Wehrle, R. Steinmetz, editors, *Peer-to-Peer systems and applications.* LNCS 3485, Springer, 2005.

[19] `http://www.haskell.org`

[20] `ftp://ftp.utcluj.ro/pub/users/gc/ispdc2011`

[21] `http://en.wikipedia.org/wiki/OpenMP`