TARDIS: A Proposal for an Agent-Oriented Program Language based on Scheme

Marcio Machado Pereira, UNICAMP, *BRAZIL*

Abstract—This paper presents a proposal of a particular agent-oriented language, called TARDIS[1]. Actually, the TARDIS is an extension of a functional language Scheme by including primitives for creating and manipulating agents. Our approach is motivated by a desire to bridge the gap between functional and agent-oriented paradigm. The syntax and semantic we developed was intended to be useful for justifying programs transformations for real languages, and for formalizing intuitive arguments and properties used by programmers.

***Index Terms*—Agent-Oriented Programming, Scheme Functional Language, Agent Programming Language, Mobile Agents, Distributed computing.**

I. INTRODUCTION

The concept of an agent, in the context of this paper, can be traced back to the early days of research into Distributed Artificial Intelligence (DAI) in the 1970s – indeed, to Carl Hewitt's concurrent Actor model [1][2]. In this model, Hewitt proposed the concept of a self-contained, interactive and concurrently-executing object which he termed 'actor'. This object had some encapsulated internal state and could respond to messages from other similar objects: an actor "is a computational agent which has a mail address and a behavior. Actors communicate by message-passing and carry out their actions concurrently" (Hewitt, C. [1]).

The meaning of the term has evolved over time in the work of Hewitt and associates. Hewitt used the term actor to describe active entities which, unlike functions, went around looking for patterns to match in order to trigger activity. This concept was later developed into the scientific community metaphor where sprites examined facts and added to them in a monotonically growing knowledge base (Kornfeld and Hewitt, cited in [2]). In Hewitt et al., the notion of actors was closer to that of an agent in Distributed Artificial Intelligence (DAI): actors have intentions, resources, contain message monitors and a scheduler. Irene Greif (cited in [2]) developed an abstract model of actors in terms of event diagrams which recorded local events at each actor and the causal relations between events.

Baker and Hewitt (cited in [2]) then formalized a set of axioms for concurrent computation which stated properties that events in actor systems must obey in order to prevent causality violations. The work in Hewitt contains the insight that the usual control structures could be represented as patterns of message passing between simple actors which had a conditional construct but no local state. It demonstrated the use of continuation passing style in actor programs, which was carried over into Scheme [3] [4] [5].

There have been a number of languages developed using the approach we follow in this paper – combining concurrency primitives with a functional language. These languages include Amber (Cardelli, 1986), Facile (Giacalone et al., 1989; Prasad et al., 1990; Thomsen et al., 1992), Concurrent ML (Reppy, 1991), Erlang (Armstrong et al., 1993), Obliq (Cardelli, 1994) and Pict (Pierce and Turner, 1994). Erlang [11] and Obliq [10] are object based languages (Erlang is essentially an actor language) while Facile, CML and Pict have process algebra concurrency primitives. Except for Facile, and to a small extent Obliq, these efforts have focused on language design, and type systems, with less attention given to semantics and equivalences.

On the other hand, some Agent specific languages, such as 3APL [13], April [6], and Go! [14], even if rich of agent-specific constructs, lack many general-purpose statements and libraries, thus needing the integration of other environments to build a complete software system.

II. THE TARDIS LANGUAGE

A. Definition Stage

Researchers in object-oriented programming have been extending the original notion of objects by incorporating one or more of the features that we have associated with agents. As a result, one has a proliferation of various extensions to objects that make them active, concurrent, distributed, reflexive,

Manuscript received June 29, 2010. This work was supported in part by the Department of Computer Engineering and Industrial Automation (DCA) at the School of Electrical and Computer Engineering (FEEC), State University of Campinas (Unicamp). Pereira, M. M. is with the State University of Campinas, Institute of Computing (e-mail: marcio.machado.pereira@gmail.com).

[1] The **TARDIS** (**T**ime **A**nd **R**elative **D**imension(s) **I**n **S**pace) is a time machine and spacecraft in the British science fiction television programme *Doctor Who*.

persistent, and real-time. However, there is no single object-oriented language that encapsulates all the above mentioned features.

Scheme is nearly an object-oriented language. This should come as no surprise, since Scheme was originally inspired by Actors, Hewitt's message-passing model of computation. Steele has described the relationship between Scheme and Actors at length [3]. We take advantage of this relationship and we try not to duplicate functionality that Scheme already provides to add full support for agent-oriented programming. Our extensions are in keeping with the spirit of Scheme: "It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions".

The primary aim of our work is to abstract the essential aspects of agents and design language aspects of agents (as well as various extensions to objects currently being attempted) within a unified framework. This paper presents an initial attempt in this direction by design of a concurrent agent-oriented language on top of Scheme.

TARDIS provides a mechanism for specifying the creation and manipulation of agents. An individual agent represents the smallest unit of coordination in the model. They are mapped as lightweight processes, that means there could be hundreds of thousands of them in a running system. Since they are an important abstraction in the language, the programmer should not consider their creation as costly. She should use them freely to model the problems at hand. An agent's behavior is described by a lambda abstraction which embodies the code to be executed when messages are received or environment changes. That is, agents are reactive as well as proactive towards the environment. The statement below creates an agent with its initial behavior.

```
(define <agent-name>
  (make-agent <behavior> ))
```

On the creation of agent a unique system generated handle will be created to access the agent anywhere in the network.

```
(let <agent-identifier>
  (spawn <agent-name>
         <initial-attributes> ))
```

This implies that no two agents of the same agency (see below), created at the same location will have the same name (a similar feature of accessing named process across anywhere in the network is available in Agent Process Interaction Language – APRIL [6]).

TARDIS agents are self-contained, concurrently interacting entities of a computing system that communicate via message passing which is asynchronous and fair. They can be dynamically created and the topology of agents systems can change dynamically. The agent model supports encapsulation and sharing, and provides a natural extension of both functional programming and object style data abstraction to concurrent open systems.

At TARDIS language, programmers can model distributed autonomous agents situated in dynamic environment that are reactive as well proactive towards the environment. For instance, agents may be organized into agencies offering certain services to other agents (these services may be realized through the execution of an associated plan, see *B. Plans and Services*):

```
(agency  <agency-name>
   (export <export-spec> )
   (import <import-spec> )
   <agency-body> )
```

The benefits of modularization within conventional languages are well known. In the model above, Agencies act like Modules, disciplining name spaces with explicit names exposure, hiding or renaming. They also offer qualified naming. These name spaces may cover services or functions, objects, agents, etc. Just as components, agencies may explicit their dependences, that is, the other agencies they require in order to work properly. Building a complete executable is done via agency fusion (like modules linking or module synthesis in case of higher-order modules). In a distributed environment this fusion takes place at execution time. During the compilation process, the compiler produces a specification file that will be shared with other Agencies. At execution time, the agency connects itself to a communication door and informs this to the service directory. In this manner, will be possible establish communication agency-agency.

The `export` subform specifies a list of exports, which name a subset of the bindings defined within or imported into the agency. An `<export-spec>` names a set of imported and locally defined bindings to be exported. In an `<export-spec>`, an `<identifier>` names a single binding defined within or imported into the agency, where the external name for the export is the same as the name of the binding within the agency.

The `import` subform specifies the imported bindings as a list of import dependencies, where each dependency specifies the subset of the agency's exports to make available within the importing agency, and the local names to use within the importing agency for each of the agency's exports.

The `<agency body>` consisting of a sequence of definitions (e.g. agents, plans, services, messages) followed by a sequence of expressions. The definitions may be both for local (unexported) and exported bindings, and the expressions are initialization expressions to be evaluated for their effects.

B. Communication Stage

Agents communicate with each other by sending messages. For instance, the following expression creates a new message with receiver <agent-identifier> and contents <message> and puts the message into the message delivery system:

```
(send <agent-identifier> <message> )
```

In TARDIS, a message can be any serializable first class value. It can be an atomic value such as a number or a symbol, or a compound value such as a list, record, or procedures, as long as it contains only serializable values.

By default, message passing in TARDIS is asynchronous. In others words, an agent can send a message whenever it likes; irrespective of the state of receiving agent. At creation time each agent is associated with a private mailbox. Then, the messages are placed in the receivers' mailbox. Synchronous and mixed mode messages can be supported also.

In addition to the agent-to-agent message passing as described above, TARDIS language supports agent-to-agency message passing. The latter will allow agents to request services on an agency basis, without having to specify a particular agent. In this case messages are routed through message-spaces, each one of which is linked to a particular agency. Message spaces will be scanned by the respective agent processes.

Although communication through message-spaces would be the preferred mode of implementation, either broadcasting or unicasting can also be implemented. In the case of broadcasting it will be necessary for a `receive` function in the receiving agent to filter the messages that are due to them.

When a message is sent by an agent to another agent or agency it is given a unique <message-identifier> and the identity of the agent which sent the message. This message is then received by the agent or agency at the other end where the message is decoded. The message is then processed by receiving agent and appropriate actions are taken.

An agent at any point in time can be in any of the following three states: `active` when it is executing a plan instance (see C. Plans and Services); `idle` when agent was suspended by any time-supervision expression or `waiting` when is waiting for a message from internal or external environment. To hold on this states, TARDIS introduced the time-supervision expressions, a fundamental way to deal with unreliable message delivery. In the following code fragment, the Agent uses the `cycle` expression to check your mailbox at cycles of elapsed time of 500 ms. When a message is arrived to mailbox the `message?` predicate becomes true and the message is retrieve by `msg` value as well as the identity of the sender is binding to `from` value:

```
(define sleep-time 500)
  …
(cycle sleep-time
  …
   (if (message? #t)
      (receive from msg))
   …
)
```

While the `receive` procedure retrieves the next available message in the agent' mailbox, sometimes it can be useful to be able to choose the message to retrieve based on a certain criteria. The selective message retrieval procedure:

```
(receive-case from
    (predicate-1 msg-1)
    (predicate-2 msg-2)
    …
    (after 10 (raise 'timeout)))
```

retrieves the first message in the mailbox which satisfies one of the predicates. If none of the messages in the mailbox satisfy the predicates, then it waits until one arrives that does or until the timeout is hit. Procedure `receive-case` specify the maximum amount of time to wait for the reception of a message as well as a value to return if this timeout is reached. In the example above, the timeout symbol will be raised as an exception. If no timeout is specified, the operation will wait forever.

The `receive` procedure can also specify such a timeout, with an `after` clause which will be selected after no message matched any of the other clauses for the given amount of time.

```
(receive from msg
    (after 10 (raise 'timeout)))
```

C. Plans and Services

An agent is deemed to exist for the purpose of accomplishing its own desires and offer certain services to other agents. Services can be defined in the scope of an agency suggesting whether is public, by the export subform, or private corresponding to internal goals of agents in the agency.

```
(define <service-name>
  (make-service  <service-body> ))
```

A service could also be pre-instantiated in expectation of a deferred invocation using the start-service function:

```
(let <service-instance>
  (start-service <service-name>
                 <list-of-arguments> ))
```

Private services instances correspond to internal goals of an agent while public services instances correspond to messages from other agents requesting to agency.

Plans are the means of performing services. A plan is identified by its <plan-name>. It specifies the <service-name> and the context in which a plan might be applicable. If the plan is applicable the goal statements are performed.

```
(plan <plan-name>
    (invoke <service-name>
        (with-context <context> ))
    (perform  <list-of-goal-statements> )
    (finalize <context> ))
```

The agent responds to the message by first selecting plans whose invocation service statement matches the service instance of the message. The invocation binding and the context binding are used to create plan instances. The agent will select one of these plan instances and start performing the goal statements. Such a selected plan instance is called an intention. At any particular instance, there can be many intentions active. Each intention is an independent thread in itself. Thus the agent as a whole is multi-threaded.

Unlike object-oriented systems the plan of an agent need not be performed sequentially from the first goal statement to the last goal statement. Any service statement in the performance of the plan results in a service instance which is sent to the agent's mail box. This process goes on till all the goal statements of the original plan are performed.

D. Mobility Stage

According to [10], mobile agents are autonomous software entities which can decide to move and relocate themselves in the network, carrying both their code and execution state. They perform tasks on behalf of a user, mobile or not. Ideally, any application using mobile agents could be programmed without them. The main interest in the use of mobile agents is to replace remote interactions with servers by local ones, in order to reduce communication costs. We think also that using mobile agents can increase expressiveness in distributed programming.

Languages and platforms for mobile agents must provide mechanisms and abstractions for:

- ✓ Concurrency and synchronization.
- ✓ Agent migration (with code, data and state) in a heterogeneous context.
- ✓ Network-level identification and localization.
- ✓ Point to point asynchronous communication.
- ✓ Security (of both agents and hosts).

Objects are good candidates for the implementation of agents, and existing mobile agents platforms are, in many cases, based on concurrent objects enhanced with mobility mechanisms (see ObjectSpace Voyager [9]). But introducing mobility in the object model is not transparent and has effects on mechanisms like synchronization and method invocation.
For example, mobility is weak in Java because it is impossible to access to thread stack values and to serialize threads. The problem of mobility degree is mainly a problem of expressiveness.

Here, we argue for actors, rather than standard objects, for mobile agent programming. When processing a message, an actor can create other actors (dynamically), communicate by asynchronous point-to-point message passing with other actors that it knows, and change its own behavior (defining the behavior for the processing of the next message). Behavior changing may be useful for agents because it provides a way for evolution and learning.

Thus, actors can be seen as active objects with the ability of changing their interface. In applications, actors as agents can be both clients and servers. Thanks to autonomy, asynchronous message passing and behavior changing, they are naturally mobile units:

- ✓ Autonomy is an important property that agents must have for mobility self-decision. The encapsulation of data and methods in the actor's private behavior (a closure) conceptually guarantees privacy and integrity. Actors encapsulate not only programs and data, but also activity. Actor systems are multi-threaded, but synchronization problems are hidden from the programmer.

- ✓ Asynchronous message passing is another important feature for mobile agents, because synchronous communications are expensive and hard to maintain in the context of wide-area or wireless networks (standard call/return is un-adapted because of latency and failures).

- ✓ Actor's behavior includes all its data and code. At behavior changing time, actor's state is fully contained in the behavior (and in the mailbox), and so, easily capturable and transferable. At this transitional moment, there is nothing more in the execution state. Movement is so delayed (only) to the end of the current message processing. Actor's mobility is based on a remote creation of an actor from the behavior intended to process the next message. Consequently, the actor moves carrying both acquired knowledge and experience. Thus, actors move but behaviors, during their execution, don't move.

Localization of moving agents is possible using a forwarding system. In TARDIS, each agent has a unique reference (like a postal address) and localization is natural by

means of a chain of forwarders. Every time the agent moves, the local agent remains after the remote creation and becomes a proxy for the agent: it receives messages for the agent and forwards them to the remote reference. Such a method allows also messages, stored in the mailbox before moving, to reach the agent after it has moved. The mobile agent reference remains valid even if moving is in progress or if the agent is remote. So, mobility is transparent for communications (code of sender agents hasn't to be modified whether the target agent is mobile or not). However, this basic protocol is known to be few efficient and fault sensitive, because of the multiple relays: several kinds of optimization can be provided. In conclusion, we can assert that enhancing agents with mobility does not involve semantics changes.

Programming mobile agents TARDIS mainly consists in defining behaviors. This is done by extending the primitives already defined and creates new ones specific to movement. In a way, the mobility mechanism (tied with behavior changing) provides strong mobility in TARDIS, since mobile agents resume remotely at the execution point where they stopped. A moved behavior contains references to agents which, in our system, do not have to be transformed. Like Obliq [10], TARDIS relies on a mechanism of network-wide lexical scoping: the main advantage is the preservation of the semantics of moved agents and the independence between computation and locality. This allows to reason upon the programs independently from the location of activities. Moving is also more efficient because referenced agents do not have to be serialized.

A `place` in TARDIS represents a virtual machine running in a physical or logical site. It can be seen as an agent server providing environment and resources for agent execution. A set of places (which can dynamically change) constitutes a domain on which applications run. If needed, it is possible to simulate distribution by creating several places on a single physical site. It is not necessary to change the code to distribute an application; the same program can be used in a local environment or in a distributed one.

To create an agent in a specific place in the network, TARDIS extends the `spawn` procedure to indicate the virtual machine where the agent will run. The variable `this-place` (binding normally to `"//localhost:"`) can be used to indicate the current place.

```
(let <agent-identifier>
   (spawn <place> | this-place
          <agent-name>
          <initial-values> ))
```

TARDIS also defines a new procedure `move!` used to move agents from their current place to a new one. We can make use of the procedure `self` that is binding to `<agent-identifier>` for a self move of the current agent.

```
(move! <place>
       <agent-identifier> | self
       (with-context <context> ))
```

III. EXAMPLE

We consider the traditional example of producers and consumers sharing a global buffer [15]. In the implementation below, we have simplified the example assuming agents know each other identities. The shared resource is accessed via the services `get` and `put`. According to standard multi-threading programming, when an agent is notified in the function `get`, it still has to check the availability of the resource. After a notification, only one agent consumes the resource. No competition is needed between agents.

```
(define producer (make-agent
    (lambda (count)
          (let loop ((n count) )
            (exec-plan put n )
            (yield)
            (loop (+ 1 n))))))

(define prod-id
    (spawn this-place producer 1))

(plan put (invoke (buffer-put! val)
  (with-context (not (full? buffer))))
  (perform
    (if (not (empty? buffer))
        (send cons-id 'resource-available))))

(define buffer-put! (make-service
   (lambda (val)
       (if (empty? buffer)
          (set! buffer (list val))
          (set-cdr!(last-pair buffer)
                   (list val))))))

(define consumer (make-agent
   (lambda ( )
      (let loop ( )
         (exec-plan get)
         (yield)
         (loop)))))

(define cons-id
   (spawn this-place consumer))

(plan get (invoke (buffer-fetch)
  (with-context
     (not (empty? buffer))))
  (perform
      (receive-case prod-id
          (eq? msg 'resource-available))))

(define buffer-fetch (make-service
    (lambda ( )
        (let ((r (car buffer)))
            (set! buffer (cdr buffer))
         r))))
```

IV. Conclusion

We describe a small set of additions to Scheme to support agent-oriented programming, including a form of mobile agent. The extensions proposed are in keeping with the spirit of the Scheme language. Our extensions mesh neatly with the underlying Scheme system. The core of this design comprises the thread and object systems used in Scheme. An important objective was that it should be flexible enough to allow the programmer to easily build and experiment this new paradigm providing higher-level primitives and a framework, so that we can share and reuse more of the design and implementation of agent-oriented programming. Another important objective was that the basic communication model should have sufficiently clean semantic properties to make it possible to write simple yet robust code on top of it. Only by attaining those two objectives can we hope to build higher layers of abstractions that are themselves clean, maintainable, and reliable.

References

[1] C. Hewitt. Viewing control structures as patterns of passing messages. Journal of Artificial Intelligence 8(3):323–364, 1977.

[2] Gul Agha. Actors: A Model of Concurrent Computation in Distributed Systems. Doctoral Dissertation. MIT Press. Cambridge, MA, USA, 1986

[3] Gerald Jay Sussman and Guy Lewis Steele, Jr.. "Scheme: An Interpreter for Extended Lambda Calculus". MIT AI Lab. AI Lab Memo AIM-349. December 1975.

[4] Harold Abelson , Gerald J. Sussman, Structure and Interpretation of Computer Programs, MIT Press, Cambridge, MA, 1996

[5] Michael Sperber , R. kent Dybvig , Matthew Flatt , Anton Van straaten, Robby Findler , Jacob Matthews, Revised[6] report on the algorithmic language scheme, Journal of Functional Programming, v.19 n.S1, p.1-301, August 2009.

[6] F. G. McCabe and Keith L. Clark. APRIL – Agent PRocess Interaction Language. Workshop on Agent Theories, Architectures and Languages. Also appears as Lecture Notes in Computer Science. Amsterdam, Netherlands, 1994, Springer Verlag.

[7] Foundation for Intelligent Physical Agents (FIPA). Fipa2000 Agent Communication Language http://www.fipa.org.

[8] A. Fuggetta, G. Picco, and G. Vigna. Understanding code mobility. IEEE Transactions on Software Enginnering, 24(5): 342-361, 1998

[9] Recursion Software, Inc. Voyager Mobile Agent Technology. http://recursionsw.com

[10] L. Cardelli. Obliq: A language with a distributed scope *Computing Systems*, 8(1):27–59, 1995.

[11] J. Armstrong, R.Virding. Concurrent programming in Erlang, second edition. Ericson Telecom Systems Lab, Sweden. Prentice Hall.

[12] H. Cejtin, S. Jagannathan, R. Kelsey. Higher-Order distributed Objects. ACM Transactions on Programming Languages and Systems 17(5): 704-739, 1995..

[13] K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Meyer Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[14] F. McCabe and K. Clark. Go! - A Multi-Paradigm Programming Language for Implementing Multi-Threaded Agents. *Annals of Mathematics and Artificial Intelligence*, 41(2-4):171–206, August 2004.

[15] Serrano, M., Boussinot, F., and Serpette, B., Scheme fair threads. In *Proceedings of the 6th ACM SIGPLAN international Conference on Principles and Practice of Declarative Programming* (Verona, Italy, August 24 - 26, 2004). PPDP '04. ACM, 203-214.

[16] Bordini, R. H., Wooldridge, M., and Hübner, J. F. 2007 *Programming Multi-Agent Systems in Agentspeak Using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons.

[17] Jean-Paul Arcangeli, Christine Maurel, Fr´ed´eric Migeon. An API for high-level software engineering of distributed and mobile applications Proceedings of the Eighth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS.01)

[18] Michael J. Wooldridge and Nicholas R. Jennings. Agent Theories, Architectures, and Languages: A Survey. Springer-Verlag 1994, pg. 1-39

[19] Shoham, Y. 1993. Agent-oriented programming. *Artif. Intell.* 60, 1 (Mar. 1993), 51-92

[20] Agha, G. A., Mason, I. A., Smith, S. F., and Talcott, C. L. 1997. A foundation for actor computation. *J. Funct. Program.* 7, 1 (Jan. 1997), 1-72.