A Comparative Evaluation of Three Mobile Languages

Zara Field
Heriot-Watt University
Edinburgh, Scotland
E-mail: zf1@macs.hw.ac.uk

P. W. Trinder
Heriot-Watt University
Edinburgh, Scotland
E-mail: trinder@macs.hw.ac.uk

André Rauber Du Bois
School of Informatics
Catholic University of Pelotas
E-mail: dubois@ucpal.tche.br

Abstract

Expressive and efficient mobile code languages are essential for the rapid construction of mobile systems. This paper provides a qualitative and quantitative comparative evaluation of three mobile code languages: Java Voyager, JoCaml and mHaskell. The languages evaluated represent a spectrum, having different programming paradigms and supporting different classes of mobility. The comparison is based on a non-trivial meeting scheduler case study that uses two common patterns of mobile computation: distributed information retrieval and multicast. Illustrated by the meeting scheduler, the languages are compared for programming model, security, language interoperability and performance on networks of 2, 4, 6 and 8 locations.

1. Introduction

This paper provides a qualitative and quantitative comparative evaluation of three mobile code languages. Java Voyager [8] is a *weak* mobile language for the Java programming platform (see section 2), with an object-orientated (OO) computational model that performs communication through method invocations. JoCaml [5] and mHaskell [3] are both research languages, where communication is performed by message passing on channels. JoCaml provides *strong* mobility with an impure functional/OO computational model based on the join-calculus, while mHaskell provides *weak* mobility, is purely high-level functional and encompasses mobility skeletons for mobility coordination [4].

The development of a non-trivial mobile application, a meeting scheduler, facilitates the practical assessment of the languages by providing a basis for comparing their performance and programmability. This application has been chosen as it encapsulates two common patterns of distributed programming, *distributed information retrieval* and *multicast*, that have the potential to benefit from code mobility. Additional features of the languages, including security and language interoperability are also assessed.

The remainder of this paper is structured as follows. Section 2 introduces code mobility and provides the classification of code mobility used in the paper. Section 3 gives a brief review of mobile code design paradigms and is followed in section 4 by an introduction to the three mobile code languages included in this comparative evaluation. Section 5 details the example application used as a basis for comparison and provides an overview of the implementation details in each of the languages. Section 6 covers the comparative evaluation and a conclusion and summary table of findings is given in section 7. Future directions for the work presented in this paper are also addressed.

2. Mobility in Mobile Code Languages

Traditionally, programs, or more generally executing computational units are statically bound to their initiating environment for the lifetime of the process. Even where code is dynamically linked, the linked code still belongs to the initiating environment. The difference with mobile code technologies is that they can allow a code segment, its state and data space to be relocated to another executing environment. The proportion of the executing unit that can be moved provides for a means of classifying the alternative mobility mechanisms [1].

Strong mobility, as found in JoCaml, enables migration of both code and execution state of an executing unit to another environment, where execution state can include execution stack, memory, program counter and state of open files. This form of mobility is facilitated by migration and remote cloning mechanisms.

Weak Mobility, as provided by Java Voyager and mHaskell, enables the transfer of code, with some optional initialisation data, but no execution state. The mechanisms that allow for this mobility allow either for the complete transfer of code to a new executing environment or simply link the code dynamically. Additionally, the direction of the code transfer can either be fetched (pulled) by an executing unit or shipping (pushed) from one unit to another, the type of code transferred can either be a fragment of code required to be linked to an already executing unit or act as stand-alone code which would instantiate a new executing unit.

3. Mobile Code Design Paradigms

Mobile code paradigms have evolved from the concept of active clients consuming the services of a passive server. As illus-

trated below, the differences lie in the location of the different components during the execution of a service.

Client-server: In this traditional design paradigm, a passive server offers a set of predefined services and statically resides at a specific location, along with all components required to execute the services. Clients, located at remote locations, actively interact with the server when requesting a service.

Remote Evaluation (REV): In this paradigm, one location has the code to perform a service but does not have the required resources, which are located at another remote location. The location with the code migrates the code to the location with the resources. The location with the resources uses its computational component to execute the code locally and uses the resources as instructed. The results are then sent back to the initiating location with a final interaction.

Code on Demand (COD): This paradigm assumes that one location contains the computational power and the required resources but lacks the code to perform the service. This location would interact with another that has the code, and the required code would be returned.

Mobile Agent (MA): The mobile agent paradigm assumes that one location has some of the resources and the code, but needs other resources to perform the service. An agent from this location migrates to another location, bringing code and possibly some results already produced by the resources located there. After it has moved to the other location it completes execution with the resources there, through local interactions. Whereas the other paradigms transfer code between the components, MA transfers a whole computational component, with state, the code and some of the resources required, to the other remote location [2].

4. Mobile Code Languages

Mobile code languages employ remote programming whereby a machine can supply and customise procedures to be performed on remote locations. Each mobile code language employs alternative computational and coordination models, primitives and abstractions when providing this mobility. The languages chosen for this comparative evaluation were chosen for their relative differences, as illustrated in table 1.

Java Voyager is an agent enhanced Object Request Broker (ORB) for Java from Recursion Software Inc. [8]. Object mobility, communication and migration are encapsulated by remote services that include interface generator, dynamic proxy generator and distributed garbage collection. Remote objects can be instantiated on separate JVM's with the remote class loading facility. The Voyager Universal Gateway automatically translates remote object messages between CORBA, RMI, COM/DCOM, SOAP and XML and new messaging protocols can be 'plugged in' once public. Object communication is provided through a variety of messaging patterns including synchronous, one-way, delayed-synchronous and asynchronous. A naming service can also be used to register and locate objects, which also supports the JNDI, CORBA and RMI naming services.

JoCaml is a mobile agent based system developed during the MOSCOVA project conducted by INRIA. The JoCaml (functional/imperative) language is based on the Join-Calculus [6] and combines primitives of Join-calculus with Objective Caml. JoCaml extends OCaml with concurrency, message-passing and message based synchronization. It facilitates the development of distributed, concurrent and mobile agent based systems wherein distribution abstractions are expressed using simple primitives taken from Join-Calculus.

Mobile Haskell (mHaskell) [3] is an extension of Haskell. It supports mobile computation by extending Concurrent Haskell with higher order communication channels, Mobile Channels (mChannels), that are synchronous and permit the communication of arbitrary Haskell values including functions, IO actions and channels. Mobility Skeletons, a small extension to mHaskell, are a library of higher-order parameterisable functions which control most, if not all aspects of coordination for three identified mobile computation patterns [4]. Unlike, JoCaml and Java Voyager, mobility skeletons provide a higher level of abstraction and eliminate the need for the programmer to provide mobility coordination algorithms.

5. Example Mobile Application: Meeting Scheduler

To compare the three mobile languages, the same meeting scheduler application has been developed in Java Voyager, JoCaml and mHaskell.

Scheduling meetings is a routine and time consuming organisational task, often complicated by communication delays and schedule conflicts. Automatic meeting schedulers are designed to reduce user intervention and thus reduce the time required to schedule a meeting. Meeting schedulers currently on the market, like Microsoft Outlook and IBM Lotus Notes 6 adopt the client-server paradigm. All processing is performed on a central machine that systematically retrieves the calendars for all peers involved and finally distributes the result, through multicast e-mail. Although this method is effective, it does not exploit the capabilities of a distributed architecture and code mobility. Scalability also becomes an issue as centralised execution on the calendar server can create bottlenecks.

Design of Mobile Automatic Meeting Scheduler

A mobile, automatic meeting scheduler is designed that exploits code mobility by distributing computation load and reducing network interactions. Users request a meeting and the scheduler automatically checks the availability of peers by sequentially migrating to their respective calendar locations, which could be located on a desktop PC or mobile device, performing availability assessments through local interactions. When the first common time is identified for all peers, a meeting allocation will be multicast that automatically updates the calendars.

In practice, a mobile agent is dispatched from a source computer to accomplish the scheduling task. The agent carries with it the computational component for the scheduling task and intermediate results as it proceeds autonomously and independently from each execution environment (calendar location) to the next. The scheduling agent is created at the location of the person intending to schedule a meeting. At this point a list of peers is provided by the user, which represents an itinerary for the scheduling agent.

The agent's behaviour is depicted in figure 1. Upon creation at host 1, the agent accesses the local calendar for a list of free times available. These intermediate free times are then transmitted with the agent to the next location (host 2), where it resumes execution and performs an intersection with the free times locally obtained at the new location. The agent traverses the list of locations performing an intersection between the locally obtained list of free times

Language	Computational Model	Mobility	Coordination Model			
			Primitives	Communication Method	Mobile Paradigms	Calculus
Java Voyager	Object-Orientated	Weak	moveTo	Method Invocation	MA, REV, COD	N/A
JoCaml	Functional/ Object-Orientated	Strong	go	Message Passing on Channels	MA, REV, COD	Join-Calculus
mHaskell	Functional	Weak	channels	Message Passing on Channels	MA, REV	N/A

Table 1. Mobile Code Languages

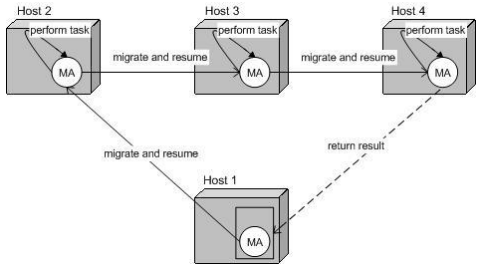

Figure 1. Distributed Information Retrieval using Mobile Agents

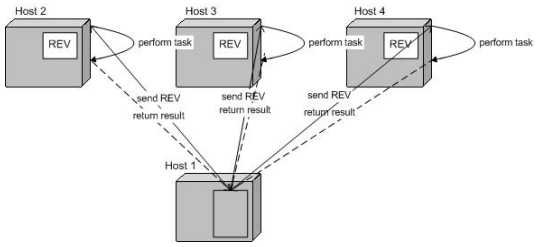

Figure 2. Multicast using Remote Evaluation

and the result of the previous intersection. At the last location, the first free time from the resulting list is sent back to the initiating program at the source machine (host 1). This is a common pattern of distributed programming known as *distributed information retrieval* (DIR).

Figure 2 shows how the initiating program, after receiving the result, *multicasts* it to all peer calendars. A synchronous multicast method provides confirmation that the calendars have been updated. This is achieved using the Remote Evaluation (REV) paradigm as the computation used to update the calendars is migrated to the specific locations and a result returned to provide confirmation.

Implementations

This section describes the implementation of the meeting scheduler in each language.

Java Voyager Implementation: This implementation, each calendar is represented by a calendar object, residing on a remote machine, that contains implementation code for the methods to retrieve free time slots. Each calendar is bound to a unique name and registered in the Java Voyager namespace for subsequent lookup. The scheduling agent is implemented through a serializable `ScheduleAgent` class that includes the methods for invoking the local methods of the calendars, an intersection method and a method for returning the result to the main program. A user provided list of names is used to lookup and create a list of calendar locations from the namespace, which then acts as the itinerary for the scheduling agent.

A mobile agent facet of the `ScheduleAgent` is obtained using the `Agent.of()` method from the Java Voyager `IAgent` interface. The Java Voyager primitive `moveTo()`, that is passed the itinerary and a user defined method `doWork`, is then called on the agent facet that forces it to move to the first location in the itinerary. The *callback* method `doWork()` is responsible for resuming the agents execution at the new location, invoking the scheduling methods, storing the intermediate results and initiating migration to the next location in the itinerary. When the agent has migrated to the last calendar object and returned the final result to the main program, it is set to autonomous, indicating that it is ready for distributed garbage collection.

Java Voyager mobile objects, an alternative to Voyager agents, were used to perform the REV multicast update. A mobility facet of a serializable `MobileUpdater` object containing the methods for updating a calendar, is obtained with the method `Mobility.of()` from the Voyager `IMobile` interface. The mobile object is then moved to the remote location with the method `moveTo()` from the `IMobile` interface, the update method invoked and the confirmation returned. The program continues to loop through the list of locations until an update object has been sent to every location.

JoCaml Implementation: In this implementation, each calendar is a standalone program residing on a remote location and contains a simple function for obtaining free times from the locally stored calendar. Both the name of the calendar and the function to obtain the free times are registered with the JoCaml name server for subsequent lookup.

The main scheduler program contains the definition of a location that is used to represent a JoCaml unit of mobility that can be moved from one calendar location to another. This mobile location acts as a container for functions used to perform the scheduling task at each calendar location, trigger the migration to the next location and finally return the result. The recursive function to perform the DIR takes as its parameters a list containing the names of the locations it is to visit and a list of free slots used in the initial intersection. Subsequently, the function takes the first name from the list of names and performs a lookup in the name server. It then migrates to this location using the JoCaml **go** primitive. Once at the new location, a lookup is performed that returns the function to retrieve the free slots for that particular location's calendar. An intersection method is performed and the result passed as a parameter to the recursive DIR call along with the remainder of the calendar locations list. Migration to the next location is then initiated.

When the mobile agent has migrated to the last calendar location and performed the intersection, the first common time and

the list of locations is passed to a JoCaml channel residing on the initial location and home of the agent. A mobile location that contains functions to update a calendar is then created that migrates to the first calendar location in the list, performs an update and returns the confirmation. Once this update has been completed, the function is called again. This remote evaluation process continues to create a mobile location and send it to the respective calendar location to update until all calendars have been updated.

mHaskell Implementation: mHaskell incorporates mobility skeletons that can be used to encode the communications required in a mobile computation (section 4). The meeting scheduler used uses first the `mfold` skeleton to identify a free slot common in all calendars with the mobile agent paradigm and finally `mmap` to multicast the result by remotely evaluating an update for each location in the list of locations that it receives as an argument [4].

In practice, each location hosts a local calendar resource that is registered with a local resource server and a function that returns all the free times available. Each location also runs a remote fork server that creates a `mChannel` with the name of the location in which it is running. This `mChannel` can later be looked up and serve as the destination location in a call to the `rfork` function that sends a computation to the `mChannel` and the `reval` function that executes a computation on a remote location and returns a result.

When the user chooses to schedule a meeting, a function is called that takes a list of locations to visit and creates a new channel `mch` that will be used to send back the result of the distributed computation. The function then uses the `mfold` skeleton and passes as its arguments the channel `mch` for returning the result of the whole computation, the scheduling function to be performed at each location, a function to intersect the results and a list that contains the locations to visit. The function then traverses each of the locations calling the scheduling function, performing the intersection and initiating migration to the next location.

Finally, when the first free time has been returned from the last location, the `mmap` skeleton is responsible for remotely evaluating the update function on every location within the list of locations.

6. Comparative Evaluation

6.1 Performance

To compare performance, the Java Voyager, JoCaml and mHaskell meeting schedulers were measured 10 times on networks of 2, 4, 6 and 8 locations. The timings obtained were for the time it took to visit all calendars (**DIR**) and the time to multicast the result (**multicast**). All three implementations were tested on the same set of Unix machines (Intel(R) Pentium(R) 4 CPU 3.2 GHz / 490MB with a 100Mb/sec. ethernet connection speed) in a quiet local network, where machine speed inconsistencies and network traffic were minimal. The current experiments ignore factors such as program size and memory consumption. Tables 2 and 3 report the median and range of execution times for the distributed information retrieval of the free times (DIR) and the multicast of the results for each language. As a measure of scalability, figures 3 and 4 plot the median distributed information retrieval (DIR) and multicast execution times against the number of locations. The following discussion is based on these tables and figures.

Jocaml Performance: JoCaml execution times for both DIR and multicast increase approximately linearly with the number of locations, with the multicast execution times being slightly more than for the DIR. As expected, a higher multicast was time required for the creation of multiple objects and the sending of multiple messages for the update method adopted. Variability in performance appears to increase super-linearly with additional locations involved. When 2 locations are involved the range is only 2ms, however when 6 locations are involved the range is over 100ms.

Locations	2	4	6	8
JoCaml				
Median.	0.058	0.132	0.221	0.299
Range	0.002	0.007	0.086	0.072
Java Voyager				
Median	0.284	0.513	0.707	0.928
Range	0.225	0.598	0.797	0.353
mHaskell				
Median	2.915	5.528	11.225	17.485
Range	0.123	0.021	0.170	0.128

Table 2. DIR Execution Times (secs.)

Locations	2	4	6	8
JoCaml				
Median.	0.058	0.132	0.221	0.299
Range	0.002	0.007	0.086	0.072
Java Voyager				
Median	0.284	0.513	0.707	0.928
Range	0.225	0.598	0.797	0.353
mHaskell				
Median	2.915	5.528	11.225	17.485
Range	0.123	0.021	0.170	0.128

Table 3. Multicast Execution Times (secs.)

Java Voyager Performance: Initial Java Voyager execution times are considerably higher than subsequent executions. This 'cold start' cost is likely to be the time taken to start the Java Voyager daemon and load the class files. Therefore, the results presented here are 'warm starts', i.e. after the initial penalty. Execution times for both DIR and multicast again increase approximately linearly with the number of locations involved, with the multicast times being slightly more than the DIR. Again, as in the JoCaml implementation, higher multicast execution times are required for the creation of the update objects, calling the update method and returning a confirmation. Variability would appear to increase super-linearly, although drops considerably with 8 locations.

mHaskell Performance: A super-linear increase in DIR execution times and an approximately linear increase in multicast is displayed, however, the multicast times are less than the DIR. This could be attributed to the additional DIR execution time required for maintaining the state of the agent before it migrates to the next location and the subsequent unpacking of the state and computation. The range of executions times is approximately uniform with increasing locations involved. This suggests that the mHaskell implementation provides the most reliable performance.

Performance Comparison

JoCaml achieves the best performance at nearly 4 to 8 times faster than Java Voyager and between 30 to 50 times faster than mHaskell. There are a number of factors that attribute to these results.

JoCaml is a compiled, strict language and hence faster than the interpreted Java Voyager implementation and the 'lazy' mHaskell implementation. Moreover, JoCaml has an optimised implementation for strong mobility. It uses OCaml primitives for serialization, provides access to the thread stack and has a number of stack management optimisations.

Java Voyager uses Java Object Serialization and RMI for mo-

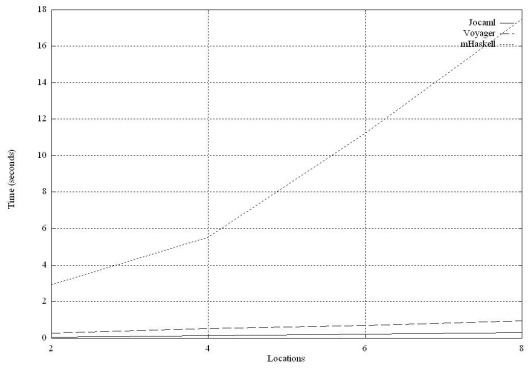

Figure 3. Median DIR using MA (secs.)

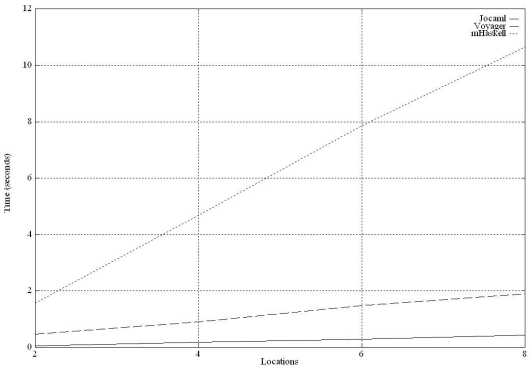

Figure 4. Median Multicast using REV (secs.)

bility, both of which are efficient and well-developed methods for marshalling, demarshalling and invoking the methods of transmitted objects. Also, Java Voyager only permits weak mobility. Modifications to the JVM that would permit accessing, saving and restoring the execution state of threads would provide for strong mobility. However, this solution would jeopardise one of the most useful features of Java, portability across platforms. Java Voyager would also appear to implement resource reuse mechanisms, which further increases performance.

mHaskell is implemented in a purely functional lazy language(Haskell), wherein even the best compilers for lazy languages are slower than the worst compilers for strict languages. mHaskell is also implemented using graph reduction, which requires double traversal of the graph. The byte code generated is communicated over TCP, then needs to be unpacked at the destination. The creation of mChannels may also attribute to the higher execution times. When a mChannel is created, a Concurrent Haskell Channel (CHC) is also created and used as the means for communication between the program and the mobile runtime system. However, possible extensions to mHaskell have been proposed that could increase its overall performance in addition to providing strong mobility [15].

Overall, the mechanisms used in both Java Voyager and JoCaml to enable mobility are more efficient than those provided in mHaskell. These mechanisms also benefit from the continual refinements designed to increase their performance. However, it terms of variability of results (i.e. the range), mHaskell is the most consistent of the three languages with Java Voyager being the least.

6.2 Programming Model

The programming model determines the ease with which one can develop a mobile code system, aided by the migration primitives provided, location and migration transparency, memory management and failure handling. The numbers of lines of code required to implement mobility and return the result of an update are also counted as a measurement of distributed transparency.

The **Java Voyager** programming model forces the use of interfaces and implementation classes, which provides for a more structured design of a system, where MA, REV and COD paradigms (section 3) are permitted. Table 4 shows that the Java Voyager meeting scheduler requires 18 lines of mobile coordination code. Migration transparency is achieved when migrating objects leave forwarder objects at the location it is about to leave, creating a forwarding chain whereby messages sent to the migrating object at an old location are then forwarded to the new location. Java Voyager benefits from the extensive Java exception facilities and also extends these with around 16 of its own that deal with migration exceptions. Java Voyagers Distributed Garbage Collection (DGC) sits on top of Java Garbage Collection. Java Voyager ORB is responsible for all remote references transparently and runs every 2 minutes updating the tables of references. When there are no references, the connection to the object is released and the object can be garbage collected. Mobile agents can also have their autonomity set to false, thus allowing for their remote garbage collection.

The **JoCaml** programming model is relatively powerful as it simultaneously supports functional, imperative and OO paradigms. It is heterogeneous, allowing agents to migrate from one platform to another. It benefits from the extended OCaml libraries for standard functions and has low level access to the system through a variety of Unix system calls. Table 4 shows that the JoCaml meeting scheduler requires 12 lines of mobile coordination code. The mobility primitives fully provide location and migration transparency. This transparency is achieved through the name server that maintains a table of object names and locations and updates it automatically when the registered object has been moved. Applets can be developed in JoCaml, permitting COD programming. JoCaml provides limited mechanisms for exception handling and recovery from partial failures. It provides an abstract model of failure and failure detection expressed in terms of locations (agents), however there is no predefined recovery mechanism and so is left under the control of the programmer. Although this does not provide for failure transparency it does offer leeway to increase the reliability of the program.

The **mHaskell** programming model uses very high-level abstractions of mobile coordination, mobility skeletons. Mobility skeletons encapsulate common patterns of mobile computation and the development of the system required only 6 lines of mobile coordination code. Location transparency is absent as the programmer needs to have advance knowledge of the location of each resource and the current implementation does not appear to allow for the migration transparency of mobile objects. mHaskell provides mechanisms for local exceptions, however these cannot be extended to a distributed setting. The mHaskell implementation can also be extended for distributed garbage collection.

6.3 Security

This section describes the security features of each of the languages and is restricted to the provision of access control mechanisms and language safety. Mobile agent security is outside the scope of this paper.

Java Voyager provides a centralised Java Voyager Security Manager, based on the Java Security Manager, which can define and implement security policies for foreign or migrating object access control. Policy files or protection domains can be used to specify what permissions are granted to specific code sources or code signed by specific persons. User authentication and access control is also available that can be extended to provide SSL (Secure Socket Layer). Java Voyager transports also provide data security and integrity with SOCKS, HTTP tunneling and Custom transports.

JoCaml has no security model or access control mechanisms. Communication over channels is unprotected and provides no guarantee of data security or integrity. However, JoCaml is safe at the language level as it employs strong static typing based on an inference system.

mHaskell provides no security model or access control mechanisms. However, it is safe at the language level as it employs static typing.

6.4 Language Interoperability

Large distributed systems are typically engineered in more than one language, and hence language interoperation is essential. The following section provides an overview of the language interoperability of Java Voyager, JoCaml and mHaskell.

The **Java Voyager** Universal ORB provides connectivity to CORBA, DCOM and RMI client and servers through a variety of protocols. However, it remains Java centric and does not interoperate with any other language.

JoCaml (OCaml) is interoperable with the C language that provides extensive libraries and it is therefore possible to interface with languages that offer a C interface. However, this interoperation between OCaml and C can raise difficulties when dealing with garbage collection, machine representation of data and the sharing of common resources and requires special consideration.

mHaskell also provides libraries for interfacing with C, through the Haskell foreign Function Interface. HaskellDirect, an IDL compiler, allows full access to Microsoft's COM library. It can also interface with the Java Native Interface (JNI) and invoke Java code, but Java cannot invoke Haskell code. Haskell can also interface with CORBA and XML and work continues to increase its interoperability.

7. Conclusion and Future Work

Three mobile languages representing a spectrum of the technologies avaliable have been comparatively evaluated for programming paradigm, security, language interoperation amd performance. The evaluation is based on implementations of a non-trivial meeting scheduler application. Table 4 suggests that Java Voyager is the most complete mobile language. Not only does it allow for the use of all three mobile code paradigms discussed, mobile applications in Java Voyager can also be fused with traditional distributed paradigms, as Java Voyager is interoperable with many existing distributed Java technologies. The inherent security of the Java language and runtime is enhanced with Java Voyagers Security Manager that allows for both efficient access control mechanisms and secure communications. Although Java Voyager only supports weak mobility effective serialization techniques are used to simulate strong mobility.

JoCaml has a concise and powerful programming model and provides the best performance. However, as a research language, these features are of lesser importance than performance.

While mHaskell mobility skeletons provide very high-level, and hence concise mobile coordination (Table 4), it is also the slowest language. This is unsurprising because, as a research language, performance has not been optimised. More significantly, mHaskell lacks important features (Table 4).

We will extend this work by further investiagting common patterns of distributed mobile programming, formalising them as *mobility* 'design patterns' [7] that will be extended with object-orientated *mobility skeletons*.

	Java Voyager	**JoCaml**	**mHaskell**
Performance Rank	2nd.	1st.	3rd.
Mobility	Weak	Strong	Weak
Calculus	N/A	Join-Calculus	N/A
Lines of Code for Mobililty	18	12	6
Mobile Paradigms Embodied	MA, REV, COD	MA, REV, COD	MA, REV
Platform Heterogenity	Yes	Yes	Yes
Agent Creation	Local/Remote	Local	Local
Agent Life Span Function	Yes	No	No
Security	Yes	No	No
Distributed Failure Handling	Yes	Partial	Possible
Remote Resource Access	Yes	Yes	Yes
Location Transparency	Good	Good	N/A
Migration Transparency	Good	Good	N/A
Communication type	Method Invocation	Message passing on channels	Message passing on channels
Language Interoperability	No (java centric)	Yes	Yes
Version	4.7	Beta version 2000	Prototype

Table 4. Language Comparison Summary

8. References

[1] G. V. Alfonso Fuggetta, Gian Pietro Picco. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[2] A. Carzaniga, G. P. Picco, and G. Vigna. Designing distributed applications with mobile code paradigms. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 22–32, New York, NY, USA, 1997. ACM Press.

[3] A. R. Du Bois, P. Trinder, and H.-W. Loidl. *m*Haskell: mobile computation in a purely functional language. *Journal of Universal Computer Science*, 11(7):1234–1254, 2005.

[4] A. R. Du Bois, P. Trinder, and H.-W. Loidl. Towards Mobility Skeletons. *Parallel Processing Letters*, 15(3):273–288, 2005.

[5] C. Fournet, F. L. Fessant, L. Maranget, and A. Schmitt. Jocaml: a language for concurrent distributed and mobile programming.

[6] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421. Springer-Verlag, 1996.

[7] E. Gamma, R. Helm, and R. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[8] R. S. Inc. http://www.recursionsw.com/voyager.htm, January 2006.