A Reflective Model for Mobile Software Objects

Ophir Holder **Israel Ben-Shaul**

Technion — Israel Institute of Technology
Department of Electrical Engineering
Technion City, Haifa, 32000, ISRAEL.
{holder@tx, issy@ee}.technion.ac.il

Abstract

Mobile software objects are autonomous computational entities that travel in large-scale and widely-distributed heterogeneous systems, and whose functionality can be attached to diverse computing environments. An object model that supports mobile objects should have special characteristics such as mutability of object's structure and semantics to facilitate adjustment to different environments, self-containment of objects to allow their migration as autonomous units, and extensive support for security. In this paper we discuss the requirements and design guidelines of such a model, and present MROM, a reflective model based on these guidelines. We also discuss MROM's implementation and present a component interoperability framework that was built on top of it, as an example application of the model.

1. Introduction and motivation

The shift from LAN-based client-server computing to global network-centric computing, involving large number of widely distributed and geographically dispersed systems and applications, has created a new need, as well as a new opportunity: the ability to transfer active data, i.e., code, over the network. Mobile code can be exploited in various ways. It can be used to overcome low-bandwidth connections by shifting interactive and other front-end computation closer to the user, as in Java Applets [1]. In the more general case, the decision as to how to split the functionality of an application between components (e.g., between a client and a server, or for balancing the load among multiple nodes) can be deferred and made on-the-fly. This is particularly useful when the type, number, identity and location of the participating nodes, as well as the bandwidths between them, is not known a priori, a typical situation in global and loosely-coupled systems. Finally, another growing family of mobile code applications involves execution of computational objects known as "agents", which exhibit some level of autonomy and/or intelligence in the form of goals, plans, itinerary. etc. (see [11]).

A crucial step towards enabling mobile software has been made with the emergence of the "write-once run-everywhere" Java technology [1], which (almost) eliminated the problem of heterogeneity in hardware and operating system platforms. However, this is only a first step. A comprehensive environment for the engineering of mobile software requires a conceptual and a technical framework: the former to enable modeling of individual objects, their interaction with other (mobile or static) objects, and their integration in a (possibly new or unknown) environment; and the latter for actual creation, transfer, host and execution of mobile objects.

As mentioned in [19] and [20], modeling distributed systems as a collection of interacting encapsulated objects with private state and behavior, is increasingly agreed as a proper and convenient approach for designing frameworks for integration of distributed autonomous and heterogeneous software components. This trend is lead by increasing standardization efforts in areas such as distributed computing, databases, and programming languages, both by official standards bodies such as ISO and ANSI, as well as by industry consortia, such as the OSF and OMG. Most object-oriented distributed frameworks incorporate a unifying common object model which is used as an agreed-upon set of abstractions that should be supported by all participating components. However, none of these models address the unique requirements of mobile objects, mostly because they were not designed for that purpose. We define the following requirements for a model that supports mobile objects:

Self-representation. This is the basic reflective property, which requires an object to be able to answer questions about itself regarding its structure and semantics. This capability is important when the host environment is not intimately familiar with the arriving object or even with its interface, in which case it must be able to interrogate the newcomer object, decide whether to invoke it, and find out how to invoke it.

Mutability. Self-representation may be viewed as a weak form of reflection. By mutability we refer to the added capability to change an object's structure and semantics, dynamically. Mutability is necessary to enable objects to *adjust* to the new context under which they are intended to operate. This is particularly important when the object may execute in different hosting environments, and/or when some negotiation is needed in order to create the initial interaction. We intentionally separate mutability from self-representation even though they are often considered collectively to comprise reflection. The reason is that in the context of mobile objects, most existing approaches are limited to self-representation and are immutable in that their structure and semantics remain fixed throughout the lifetime of the object. Mutability also raises other issues regarding class-instance relationships, addressed in Section 4.

Self-containment. This requirement is related to autonomy. Since a mobile object is intended to execute in different locations, it must contain all the functionality in itself and minimize dependence in the host environment. For example, a long-lived persistent mobile object should contain its own persistence scheme and be able to write itself to disk on a space allocated for it by the host environment, as well as read itself into memory following some bootstrap procedure initiated by the host environment. Self containment also implies *decentralization*, since the overall computation and state of the system is necessarily distributed in the autonomous objects, hence not centralized.

Security. Given a very large universe of objects in which mobile objects may execute in domains with varying levels of trust, a security model (including policies, not only mechanisms) must be associated with the object model. Note that security concerns are mutual. That is, not only should the host environment be able to restrict the operation of the mobile object, the mobile object should also be able to restrict access by the host environment (particularly with respect to access to self-changing operations) as in encapsulation. This observation suggests coupling encapsulation with security.

Weak Typing. Mutability implies that no long-term guarantee regarding the exact structure and semantics of an object should be made in advance, but rather should be left to be finalized in runtime, as done in other high level programming languages such as Scheme and Prolog. Another reason for weakly constraining types is the heterogeneity of the objects and the bottom-up style of the creation of mobile-based applications, which implies that the object model should support generic coercion to facilitate the high level of abstraction (e.g., to transform a value that is represented as HTML text into an integer, when arithmetic operation should be performed on that value).

Identity and Naming. As mentioned before, the model should not be limited by the number, size, or geographical dispersion of the objects in the system. Thus, there should be built-in decentralized mechanisms for assigning distinct names for objects.

Advanced Features. In this category we include issues of persistence, synchronization mechanisms to allow implementation of concurrent programming models, and atomicity to facilitate consistent computations.

This paper presents a model for mobile objects, focusing mainly on the first three requirements, namely self-representation, mutability and self-containment. Persistence, security and naming are the subjects of other papers ([16], [17]). Section 2 compares our work to other object models; Section 3 presents the actual model; Section 4 highlights issues related to the implementation of the model; Section 5 describes the use of the implemented object model in HADAS, a framework that supports mobile objects and was built using the object model. Section 6 summarizes our contributions.

2. Related work

While not designed to support mobile software objects in the first place, object models of popular distributed computing frameworks seem to be the most widely available infrastructures for utilization of such objects.

In CORBA [21], [23] the Dynamic Invocation Interface (DII) mechanism can be regarded as supporting some level of reflection. DII allows dynamic lookup of a desired interface in an interface repository, and getting all the required information from the repository so that a request on an object that implements the interface can be built. This feature, along with the ability to dynamically change the repository, allows dynamic changes in the meaning of a certain interface. CORBA does not limit an interface to be implemented only by one object, which can be viewed as providing several semantics to the same interface. Nevertheless, reflection is not explicitly supported by CORBA and the core object semantics, such as the invocation mechanism, is not subject to any manipulations. Security is also not explicitly supported by the model, but rather left for object implementers.

The Distributed Component Object Model (DCOM), which serves as the base for Microsoft's component computing framework (OLE, or ActiveX), incorporates some level of reflection through the use of *Interfaces* [7], [6], [21]. An interface in DCOM is a set of functions bounded to a certain object which implements them. Each object may introduce several interfaces and a user may query any one of them using the `QueryInterface` function, which belongs to a default interface supported by every object. `QueryInterface` can be regarded as a reflective method, since it practically changes the semantics of the object as being seen by its user. However, while an object's interface can be changed in runtime (e.g., a new interface can be added) object's implementation can not. Rather, such changes require recompilation of the object's source code. Furthermore, there is no notion of a fixed behavior for an object since objects are entities unknown to their users (only the interfaces are known). Thus, an object that supports a certain interface in a particular time can be changed and appear later without support for that interface, introducing inconsistency. As with CORBA, the issue of security is not explicitly supported by the model, but left for object implementation.

Another noteworthy object model is that of the Java language [1]. Java combined with its Remote Method Invocation (RMI) package [26] forms a framework for construction of distributed systems, based on the Java object model. The basic Java object model has no explicit support for reflection. However, some level of reflection is supported in JDK 1.1 [14] as part of the API. Though supplying facilities for querying object's structure, such as to examine its methods and their signatures, this API does not support mutability, e.g., it does not allow operations on existing objects that may change their semantics. Java was designed with great emphasis on security, and this issue is tightly coupled with its object model through the use of the `SecurityManager` class. The entities subject to security restrictions are system resources such as files, sockets, etc. Our approach is to regard methods and data-items as subject to those restrictions and apply security checks on one action only — method invocation.

Extensive research has been conducted on reflection in object-oriented languages. The primary goal is to achieve maximum semantic flexibility combined with clean design. This objective is very hard to achieve while keeping the system simple and efficient. Indeed, as analyzed in [12], complex architectures where reflective functionality is assigned to special meta-objects, which themselves can have meta-objects that control their own behavior, is common in these languages. In addition, as justified in [18], implementation using a meta-circular interpreter architecture [2], usually on top of LISP, is common to most of these languages (e.g., 3-KRS [18], ObjVlisp [10] and CLOS [15]). This approach also imposes a significant price on performance. Open C++ [9] and Choices [22], on the other hand, presents a simpler and relatively efficient approach which adds some level of reflection to C++ for particular applications (distributed programming and operating system, respectively). Since our focus is in facilitating a distributed framework, as opposed to reflection in itself, we adopt the Open C++ and Choices approach of relieving flexibility and robustness, while preferring to include reflective features only where we think they are required.

3. The mutable reflective object model (MROM)

Several key issues are involved in designing an object model that fulfills the unique requirements of mobile objects, as outlined in Section 1. The most important issue concerns the level of mutability (subsuming self-representation which is needed for any level of mutability). That is, to what degree can an object's structure and semantics be altered. Note that we refer here to object-based changes, as opposed to changes made to a class which apply to all of its instances, as in class evolution (e.g., [4]). This distinction is important because object-level mutability implies that an object may be modified in such a way that it does not follow the structure of its original class, thereby weakening type-safety and introducing potential runtime errors due to mismatch between the caller's formal expectation of an object's structure (both data and methods), and the actual structure (formal and actual are analogous to formal vs. actual parameters). Moreover, unrestricted mutability provides no guarantee as to what functionality or data an object will have at a certain time, thus reducing the usability of such an object. In particular, no security measures can be enforced on such an object, if it can change them on its own. Finally, structural mutability bears some price on performance, because it implies that technically there must be an internal mechanism to lookup the location of an item before accessing it (e.g., via a pair of `get` and `set` operations), whereas in static structures the location is determined at compile time as a fixed offset. On the other hand, as motivated earlier, mutability of structure and functionality is essential in order to adjust to the specific circumstances and constraints of the context under which the object has to operate.

Our approach balances these conflicting requirements, by splitting an object into two sections, *fixed* and *extensible* . Data items and methods defined in the fixed section of the object are treated as conventional items, and may not be changed during the object's lifetime (which may be arbitrary long, particularly if the object is persistent). This portion of the object should be used to store its fundamental state and behavior that comprises the core functionality of the object. In contrast, the extensible section comprises the mutable portion of the object through which object's structure and behavior can be changed, and into which new items (data or methods) can be added, removed or changed. This portion can be utilized to attach (detach) data and functionality on-the-fly. For example, one methodology of using extensible items in the design of certain kind of mobile objects (which we are actually exploring in the HADAS project) may be to place interface-related functionality in the extensible section, which then can be adjusted to the interface requirements of the object with which it interacts. Finally, an important aspect of the distinction between the fixed and the extensible sections of an object is with respect to object specialization, or inheritance. The fixed section represents the stable portion of the object whose structure and behavior is always guaranteed to exist, thus can be used for specialization by the programmer. Items of the extensible portion, on the other hand, are subject to changes in runtime, thus can not be counted on to have any certain semantics at any given time.

A related but separate issue to consider is *how* mutability is achieved. Here too, there are two main alternatives: external and internal. The former refers to changes which are made by the system, i.e., outside the object, whereas the latter refers to changes which are made by the object itself. Note that this is orthogonal to the aspect of who initiates the change: in either case, the initiator may be the object itself or an external requester. Our (significant) choice here is to adopt the *reflective* approach, namely, to allow the object to change its own structure. The main reason for this choice is to satisfy self-containment (and autonomy). Since an object may move and operate in different territories, it must be self-contained, particularly with respect to controlling its own structure and behavior. Technically, this means that each object must contain meta-methods for the manipulation of the structure and semantics of itself, and for method invocation. Self containment implies that we refrain from separating the meta-methods in a distinct meta-object, as in [18], and bundle them inside the object.

More specifically, our reflective facilities include the following meta-methods:

- `getDataItem/setDataItem/addDataItem /deleteDataItem` — These operations are used to examine and manipulate the data elements of an object, but not their values (which are accessed using ordinary `get` and `set`).Thus, they are only applicable on items which are defined as extensible. `add` and `delete` are straightforward, adding new data members and deleting existing ones, respectively. `getDataItem` returns a description of the data item and a handle that can be used by `setDataItem` to change its properties such as security access or encapsulation, name, or their dynamic type, if any.
- `getMethod/setMethod/addMethod/ deleteMethod` — These are the analogous operations on methods, with similar semantics. There is no conceptual reason for creating two sets of commands; the sole reason is to avoid name conflicts between data items and methods.
- `invoke` — This is the most important meta-method. It implements method invocation and is used to invoke any method of the object, *including meta-methods*. The only exception is `invoke` itself, which may or may not be invoked by a copy of itself, as explained below.

The combination of reflection and mutability raises another interesting possibility: to enable a class designer to make meta-methods also extensible, i.e., mutable. While *meta-mutability* (i.e., the ability to change the object-changing methods) might seem far-fetched for the domain of mobile software objects, where most current reflective approaches are still restricted to introspection only, this approach may indeed have important practical uses. For example, it may be desirable to change the `invoke` method (using `setMethod`) in such a way that an object contacts another (possibly remote) “approval” object prior to the actual invocation, or alternatively it contacts a “charging” object from which the invoked object was rented (following Yourdon's “code renting” concept [28]). To enable such flexibility, it should be possible to place meta-methods, including `invoke`, in the extensible section. This powerful approach introduces the problem of circularity when applied to invocation, in that the (meta)invoke method has to be itself invoked. We solve this problem by adding an extra level of indirection, and by implementing a “primitive”, level 0 invocation mechanism, with identical semantics as the basic meta-`invoke` method (the base invocation semantics are explained below). If no modifications are desirable, only level 0 is used. To modify the invocation, an extra level 1 invocation is created, which is invoked form level 0. In fact, nothing in the model prevents the creation of arbitrary levels of invocation, although we have not encountered yet practical situations that demanded more than two levels. This technique is analogous to implementing a reflective language using a meta-circular interpreter in that semantic consistency between the self-representation of the system and the system itself is being kept by implementing a copy of the meta-behavior in another non-reflective language and implementing the meta-behavior in terms of that copy, as discussed in [18] and implemented in CLOS [15]. We now turn to the explanation of the semantics of the core level 0 method invocation mechanism, which may be viewed in the general case as the stopping condition of the recursive invocation mechanism.

3.1 Level 0 invocation

This mechanism incorporates built-in support in two main aspects which are particularly relevant to our application domain: *wrapping*, and *security*. Wrapping refers to support for adjusting and/or integrating a computational object into the (new) environment under which it operates. For example, if the method involves operating a remote CORBA object, a preparation operation should be invoked to generate and install that object's stub (a fairly complex operation that involves local compilation). Another domain in which wrapping is common is software engineering environments (e.g., Oz [5], Field [25]) and workflow management systems [13], which typically incorporate mechanisms to integrate external tools into the environment. To facilitate wrapping, each method can be wrapped with *pre*- and *post*-procedures, which are called before and after the invocation of the body of the method, respectively. These procedures can be attached to the method dynamically (by invoking the `setMethod` meta-method). Another use of pre- and post-procedures is as assertions on the method, similarly to the approach taken in [24] to add assertions to C++. To enable such behavior, both operations always return a boolean value. A False return value from pre-procedure prevents from invoking the body of the method and a False from a post-procedure raises an exception.

Pre- and post-procedures can also be used as a convenient reflection facility. For example, one way to implement the charging mechanism mentioned earlier is to add level 1 (meta)invoke method with its pre-procedure performing the required charging. Since the pre-procedure is on the `invoke` method itself, it applies to the invocation of all methods in the object, as opposed to specific methods. The reader may wonder why not change the pre- and post-procedures of level 0 invocation, which avoids the need to create level 1 invocation. The answer is, that even though level 0 possess the same semantics as level 1 (before it is changed), its representation is not visible and non-reflective, is not accommodated for change, and can be implemented in a more efficient way. Changing level 0 invocation would be analogous to allowing LISP programmers to change the semantics of the language by modifying the base interpreter, as opposed to changing the meta-circular LISP interpreter written in LISP.

The second built-in feature of the base invocation mechanism is inherent support for security. Here we take a unique approach: coupling security with encapsulation. By that we mean that controlled access to each data-item or method should serve both for visibility purposes — as with ordinary object-oriented programming languages — as well as for ensuring legitimacy of getting and setting data-items and of invoking methods, operations whose execution may be restricted for a specific group of other objects. Since the scope of participants in a computation is the whole universe, including ones from trusted as well as from untrusted domains, the granularity of access availability should be the single object, as opposed to classified as either public, private, or other inheritance-related visibility categories (e.g., protected). Thus, each method has associated with it an *access control list (ACL)* that specifies which other objects can access it. for more details on the security mechanism, see [16], [17].

Altogether, the basic method invocation mechanism consists of three phases:

1. Lookup — locate and fetch a method's handle.
2. Match — match security information.
3. Apply — invoke the operation on the method, consisting of the following phases:

   3.1. Pre-proc — invoke the pre-procedure.
   3.2. Body — transfer control to body of method.
   3.3. Post-proc — invoke the post-procedure.

Figure 1 (on the next page) illustrates the invocation mechanism combined with its own meta-mutability. It depicts a two-level invocation of the method `Mfoo` of object `Obar`, where `meta_invoke` is a modified invocation mechanism. Notice the process of passing arguments, where the method `Mfoo` is sent as a parameter to `meta_invoke`, and is later invoked by it (following level 0 invocation).

To summarize, reflection and invocation are separate yet intertwined aspects of our model: the invocation mechanism facilitates reflection (by means of pre- and post-procedures), and the reflection mechanism facilitates modification of the invocation mechanism. It is up to the object designer to choose none, either one, or both of these mechanisms to create and modify a highly adjustable yet internally consistent and secure object.

4. Implementation of MROM

MROM is implemented in Java, the obvious choice given its support for security and platform-neutral object-code. However, while playing a major role as an enabling technology, Java is used only as an underlying implementation system with no linkage between its own object model and MROM's. Both data-items and methods are implemented as Java classes. The data-item class holds the actual MROM (untyped) datum as a Java data-member and the method class holds MROM method components (body, pre- and post-procedures) as Java methods.

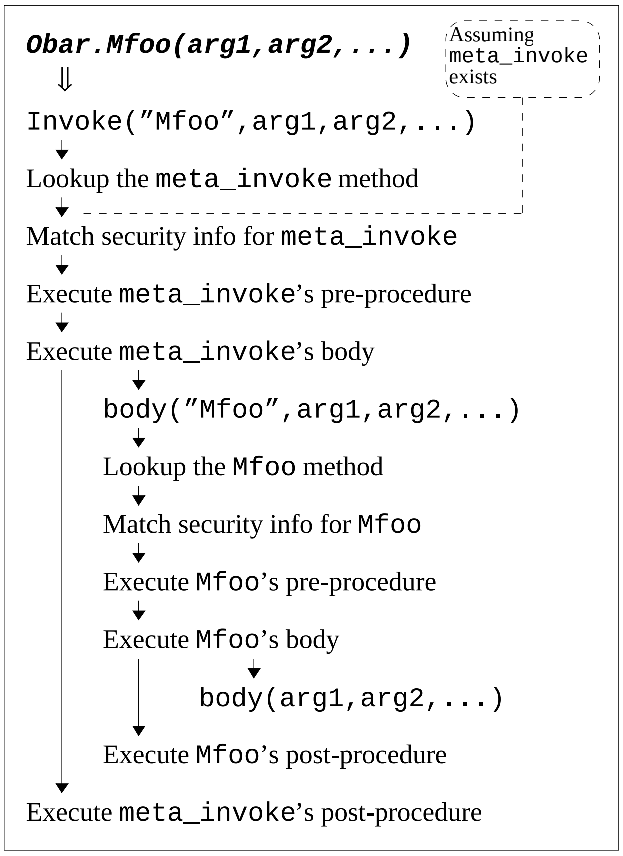
Obar.Mfoo(arg1,arg2,...)
Assuming meta_invoke exists
Invoke(”Mfoo”,arg1,arg2,...)
Lookup the meta_invoke method
Match security info for meta_invoke
Execute meta_invoke’s pre-procedure
Execute meta_invoke’s body
body(”Mfoo”,arg1,arg2,...)
Lookup the Mfoo method
Match security info for Mfoo
Execute Mfoo’s pre-procedure
Execute Mfoo’s body
body(arg1,arg2,...)
Execute Mfoo’s post-procedure
Execute meta_invoke’s post-procedure

Figure 1 - Two Levels Invocation

Following the weak typing requirement, MROM methods receive an arbitrary number of untyped objects as parameters. This is realized by passing an array of Java objects as a single parameter to the body of all methods.

The fixed and extensible portions of MROM objects are implemented using four Java objects called *item containers*. An item container is a set of name-and-value pairs, where the value is either one of the object's data-items or one of its methods. The extensible portion of an MROM object consists of two extensible containers, whose pairs can be added, removed and their value can be replaced in runtime. The fixed portion consists of two containers on which none of the previous manipulations are available.

The MROM objects themselves are also implemented as Java classes that hold the above four containers. Level 0 invocation is implemented as a Java method of the MROM object class, which is used to perform all (MROM) invocations.

Static (not in run-time) specialization of MROM objects is achieved using Java sub-classing. Copying the containers of the super-class to the sub-class, as well as adding items (recall that adding items to the fixed containers is impossible in runtime), are done in the sub-class constructor. Since implemented as Java classes, static specialization of MROM data-items and methods is also possible through sub-classing. Note that the mutable nature of MROM objects provides means of dynamic (in-runtime) specialization by the ability to add new fields and to change existing fields, which gives an effect similar to that of inheritance in prototype-based languages (e.g., Self [27] and Cecil [8]).

5. The HADAS framework

HADAS (Heterogeneous, Autonomous, Distributed Abstraction System) [3] is an interoperability framework that is aimed at facilitating the construction of network-centric applications by means of *interoperability programming*. Specifically, HADAS provides support in four levels of interoperability: (i) *Integration* of pre-existing components into the framework as entities that can be referenced and accessed by other components, as well as be able to reference other integrated components. (ii) *Communication* level that includes agreements over low-level protocols, marshaling schemes, component identification and location mechanisms, as well as middleware solutions for bridging and/or mediating syntactic mismatches in data formats, argument passing, etc. (iii) *Configuration* level that includes notations and utilities for establishing (dynamic) agreements between components as well as managing the (dynamic) bindings between them, separately from the programming of the components themselves. (iv) *Coordination,* the highest level, is concerned with semantic interoperability, allowing to specify control- and data-flow between (integrated, interconnected and configured) components.

HADAS incorporates MROM as its base object model both for abstracting the application components themselves, as well as for the system’s internal structure. Each logical “site” in HADAS is represented as an InterOperability Object (IOO). This object serves as a container of both a collection of components and of multi-site InterOperability Programs, and as a primary contact point for other IOOs for components interaction. Depicted in Figure 2, the state of an IOO includes the following objects: (i) *Home*: A container whose data-items are APplication Objects (APOs) that encapsulate real applications, both legacy and native-HADAS. (ii) *Vicinity*: A container whose data-items are objects called *IOO Ambassadors*, each representing a remote IOO with which a cooperation agreement has been established. (iii) *Interop*: A (methods) container whose methods are coordination-level programs (not shown in the figure).

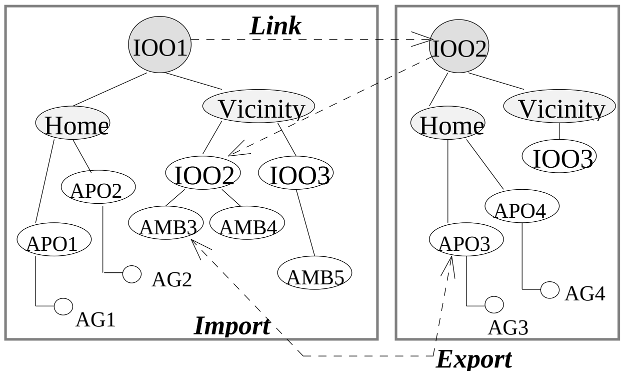

Figure 2 - HADAS External View

APOs also have *Ambassadors* (AMBs in the figure) which reside in remote sites. An Ambassador is an object that has been instantiated in the origin APO and has been deployed in a “foreign (IOO) territory”, but is owned and maintained by its origin APO. Each Ambassador thus has exactly one origin and is hosted by exactly one IOO.

Establishing cooperation agreements between two components involves binding and configuration activities. The basic methods used to create agreements are:

- *Link* — An IOO method that establishes connection between two IOOs. A successful invocation of this method installs an Ambassador of another IOO, at the Vicinity component of the IOO whose Link has been invoked. This operation is a prerequisite for any further cooperation between the two IOOs.
- *Import/Export* — The purpose of these operations is to establish the connection between a remote APO and a local IOO, for later invocation by an interoperability program. An Import operation at the requesting IOO is handled by an Export operation at the receiving IOO. Export verifies that the requested APO is accessible to the requesting IOO, instantiates the proper APO Ambassador object, and sends it to the requesting IOO. When the Ambassador arrives (as data) the importing IOO unpacks it, passes to it an installation context and invokes the Ambassador, which in turn installs itself in the new environment.

Ambassadors, which play an important role in the HADAS architecture, are an example of mobile software objects that exploit MROM’s capabilities. In particular, object mutability can be used to dynamically determine how to split a component’s functionality between the APO and the Ambassador. The dynamic migration of functionality (methods) and data from the APO to its ambassador and vice versa, can be done using the meta-methods. Furthermore, updates in APO’s functionality can be done dynamically without interference with on-going computations that need the APO, by adding methods and data items to the APO and its Ambassador on the fly. Such dynamic update is possible, of course, only in the extensible sections of the Ambassador. Any behavior and state of the Ambassador that has to remain untouched in order to maintain its consistency is defined in the fixed section of the object.

The natural connection between security and encapsulation in a distributed framework can be exemplified in the relationship between IOOs and Ambassadors. As stated previously, Ambassador’s semantics should be updated only by its origin. This implies that its meta-methods should be invisible to the host IOO as a matter of programming policy (encapsulation issue), and should not be invoked by that IOO to protect the Ambassador and its origin from malicious intervening (security issue). The opposite direction also exemplifies this point. The Ambassador, as a servant for its host IOO, should not see most of IOO’s methods (encapsulation) and as an invading entity, should not be granted permission to manipulate the IOO (security). MROM’s combination of security and encapsulation facilitates this duality.

As an example for the use of dynamic changes in semantics of invocation, consider a database APO whose methods return employees information. This APO may include a method that changes the invocation mechanism in all its Ambassadors such that upon every invocation, a message stating that the database is down for maintenance is being echoed. Before shutting-down the database, its administrator can invoke this method to update the Ambassadors, such that users at remote sites can have instant meaningful results for their queries, instead of long waiting and misunderstood error messages. Note how autonomy of both the database and its remote clients is being preserved. The activity of database shut-down should not be approved in advance by the administrator, and applications that uses query results can continue to work since meaningful responses are being returned.

HADAS is fully implemented in Java (50K lines of code) and uses some of its advanced features such as RMI and serialization. For more details on HADAS and its architecture see [3], [16], [17].

6. Conclusions and future work

New opportunities for construction of very-large-scale distributed systems are presenting new challenges in distributed software design. We are evidencing a shift from centralized, top-down construction, which is adequate for systems composed of homogeneous standard-based components, to decentralized, highly dynamic, bottom-up construction, (re)using heterogeneous and autonomous components. The concept of a mobile software object, which can migrate in a wide-scale distributed system, carry its functionality to foreign environments, allow those environments to discover dynamically how to interact with it, and adjust its own structure and behavior to fit the particular needs of the host, is essential to realize this computing paradigm shift.

The main contribution of this work is in identifying the key design guidelines for an object model that facilitates mobile software objects, in designing a model (MROM) based on these guidelines, and in implementing a framework for component interoperability as an application of the model. The guidelines identified are: (i) Mutability through reflection, which facilitates dynamic changes in object semantics while not requiring the system to rebuild itself, thereby also preserving component autonomy. (ii) Security coupled with encapsulation, thus becoming an element of the model instead of an ad-hoc add-on. And (iii) splitting objects into fixed and extensible sections, to enable dynamic changes while preserving core functionality immutable, thus balancing between dynamic evolution and the fixed base of functionality needed by all extenders.

We are currently evaluating performance overhead incurred by our reflective model and exploring various optimizations since the issue of efficient implementation of MROM is important for its practical deployment. Other future directions include the development of methodologies, tools and protocols to aid in the design and implementation of applications, and building more practical applications involving mobile objects. One step further would be to build a programming language around MROM that facilitates "mobile programming".

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[2] H. Abelson, G. Sussman, J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[3] I. Ben Shaul, A. Cohen, O. Holder and B. Lavva. HADAS: A network-centric framework for interoperability programming. Technical Report EE Pub No. 1079, Technion, Department of Electrical Engineering, January 1997.

[4] J. Banerjee and W. Kim. Semantics and implementation of schema evolution in object-oriented databases. *Proceedings of the ACM SIGMOD Annual Conference on the Management of Data*, pp. 311-322, May 1987.

[5] I. Ben-Shaul and G. Kaiser. *A Paradigm for Decentralized Process Modeling*. Kluwer Academic Publishers, 1995.

[6] N. Brown and C. Kindel. Distributed Component Object Model Protocol - DCOM/1.0. Internet Draft, May 1996. http://www.microsoft.com/oledev/olecom/dcomspec.txt

[7] K. Brockschmidt. *Inside OLE*. Microsoft Press, 1995.

[8] C. Chambers. Object-oriented multi-methods in Cecil. *ECOOP '92 Conference Proceedings*, pp. 33-56, 1992.

[9] S. Chiba and T. Masuda. Designing an extensible distributed language with a meta-level architecture. *ECOOP '93 Conference Proceedings*, pp. 482-501, 1993.

[10] P. Cointe. MetaClasses are first classes: the ObjVlisp model. *OOPSLA '87 Conference Proceedings*, Published as *SIGPLAN Notices*, 22(12):156-165, October 1987.

[11] *First International Conference on Autonomous Agents (Agents '97)*, Marina del Rey, California, February 5, 1997.

[12] J. Ferber. Computational reflection in class based object oriented languages. *OOPSLA '89 Conference Proceedings*, *SIGPLAN Notices*, 24(10):317-326, October 1989.

[13] D. Georgakopoulos, M. Hornick and A. Seth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3:119-153, 1995.

[14] Java Core Reflection. Available as: http://www.javasoft.com/products/jdk/1.1/docs/guide/reflection/index.html

[15] G. Kiczales, J. des Rivieres and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1992.

[16] B. Lavva, O. Holder and I. Ben Shaul. Object management for network-centric systems with mobile objects. Technical Report EE Pub No. 1078, Technion, Department of Electrical Engineering, January 1997.

[17] B. Lavva, O. Holder and I. Ben Shaul. Persistence and security support for distributed systems with mobile software objects. Position Paper. Accepted for publication in the *3rd International Workshop on Next Generation Information Technologies and Systems (NGITS'97)*, June 1997.

[18] P. Maes. Concepts and experiments in computational reflection. *OOPSLA '87 Conference Proceedings*, published as *SIGPLAN Notices*, 22(12):147-155, December 1987.

[19] F. Manola. Metaobject protocol concepts for a 'RISC' object model. Technical Report TR-0244-12-93-165, GTE Laboratories Incorporated, December 1993.

[20] F. Manola. Interoperability issues in large-scale distributed object systems. *ACM Computing Surveys*, 27(2):268-270, June 1995.

[21] F. Manola. X3H7 object model feature matrix. Technical Report X3H7-93-007v10, GTE Laboratories Incorporated, February 1995. Also available as: http://info.gte.com/ftp/doc/activities/x3h7.html

[22] P. Madany, P. Kougiouris, N. Islam, and R.H. Campbell. Practical examples of reification and reflection in C++. *Proceedings of the International Workshop on New Models for Software Architecture*, November 1992.

[23] J. Nicol, C. Wilkes and F. Manola. Object orientation in heterogeneous distributed computing systems. *Computer*, 26(6):57-67, June 1993.

[24] S. Porat and P. Fertig. Class assertions in C++. *Journal of Object Oriented Programming*, 8(2):30-37, May 1995.

[25] S. Reiss. Connecting tools using message passing in the FIELD environment. *IEEE Software*, 7(4):57-66, July 1990.

[26] Java Remote Method Invocation. Available as: http://www.javasoft.com:80/products/jdk/1.1/docs/guide/rmi/index.html.

[27] D. Ungar and R. B. Smith. Self: the power of simplicity. *OOPSLA '87 Conference Proceedings*, *SIGPLAN Notices*, 22(12):227-242, December 1987.

[28] E. Yourdon. Java, the Web, and software development. *Computer*, 29(8):25-30, August 1996.