7

A Framework for High Assurance Security of Distributed Objects

John Hale, Jody Threet and Sujeet Shenoi
Department of Computer Science
University of Tulsa, Tulsa, Oklahoma 74104-3189, USA
{hale, threet, sujeet}@euler.mcs.utulsa.edu

Abstract

High assurance security is difficult to achieve in distributed computer systems and databases because of their complexity, non-determinism and inherent heterogeneity. The practical application of formal methods is the key to high assurance security in open, distributed environments. This paper proposes the use of formal methods and a special layered architecture to achieve secure interoperation of heterogeneous distributed objects. The foundation is provided by ROC, a process calculus tailored for concurrent objects. Lying above ROC in the layered architecture is a meta-object model for creating object models with various programming constructs, mega-programming facilities and security mechanisms. Successive layers of the architecture represent more sophisticated toolkits for modeling distributed objects. Since each layer inherits ROC's formal foundation, it automatically has an unambiguous semantics and supports verification.

Keywords

Distributed objects, high assurance security, formal methods, process calculus

1 INTRODUCTION

The rapid growth of the Internet and the emergence of the World Wide Web as a computing paradigm have brought distributed systems into the mainstream of personal computing. The massive interconnectivity and interoperability of distributed computing resources offers tremendous benefits, but renders them more vulnerable to security threats. Unfortunately, the complexity, non-determinism and inherent heterogeneity of distributed systems makes them extremely difficult to secure. Even now, security in most large computer networks is a confusing patchwork of diverse models and *ad hoc* mechanisms and policies. One solution is to deliver applications with high assurance that they can operate securely in open, distributed environments.

The practical application of formal methods is the key to high assurance computing. Formal methods have been applied to centralized computer systems and traditional programming languages with some success (Diller, 1990; Wing, 1990; Hinchey and Bowen, 1995). Unambiguous

P. Samarati et al. (eds.), *Database Security*

formal semantics for these systems created by applying formal methods provides the basis for system/application verification. The view of a security policy as a logical proposition leads to the consideration of verifiably secure computer systems. Formal models of computer security can provide precise semantics for security models, mechanisms and policies. These semantics and their accompanying verification properties are indispensible to realizing the goal of verifiably secure heterogeneous distributed systems.

While many research efforts have applied formal methods to high assurance computing (Diller, 1990; Wing, 1990; Hinchey and Bowen, 1995), a practical application of formal methods to heterogeneous distributed system verification remains elusive. The Meta Object Operating System Environment (MOOSE) described in this paper employs a special layered architecture to achieve high assurance secure interoperation of distributed objects. Objects provide a clean, realistic model of persistent entities with complex behavior found in most heterogeneous distributed systems. Several architectures, most notably CORBA (Object Management Group, 1991; Mowbray and Zahavi, 1995) and DCE (Open Systems Foundation, 1992; Rosenberry, Kenney and Fisher, 1993), have been proposed as standards for distributed object management systems. However, unlike MOOSE, they lack the formal foundation necessary to verify system security and other critical properties of high assurance systems.

The foundation for MOOSE is provided by the Robust Object Calculus (ROC), a process calculus tailored to modeling distributed object systems. Upon ROC rests the Meta-Object Model (MOM), an ACTORS-like architecture (Agha, 1986) for building concurrent/distributed systems. Any object language or model constructed with MOM inherits ROC's formal semantics which provides a basis for system verification. Existing languages such as Common Lisp Object System, C++ or Java can be given ROC semantics, while object code can be accompanied by abstract ROC models to maximize interoperability. Object architectures constructed using MOM can contain powerful programming constructs, mega-programming facilities and security mechanisms. Successive layers of the architecture represent more sophisticated toolkits for modeling distributed objects. Since these layers also inherit ROC's formal foundation, they have unambiguous semantics and support verification.

Layered architectures have been used by several researchers to construct verification systems for programming languages and distributed systems (Bevier, *et al.*, 1989; Alves-Foss and Levitt, 1991; Zhang, *et al.*, 1994, 1995). The Silo Project at the University of California-Davis has applied a layered architecture to the formal verification of secure distributed systems and applications (Zhang, *et al.*, 1994, 1995). This work advances Silo by employing a primitive process calculus (ROC) for concurrent objects as a foundation for the semantics hierarchy. Using ROC as the execution model for distributed systems has several advantages. ROC's formal operational semantics facilitates a mechanization (deep embedding) into a more expressive mathematical system, e.g., the higher order logic (HOL) (Gordon and Melham, 1993). ROC can be used as a semantic algebra for creating denotational semantics for concurrent object-oriented programming languages. Applications written in these languages inherit the formal operational semantics of ROC, allowing reuse of ROC's HOL theorems. These features make high assurance security attainable in open, distributed systems.

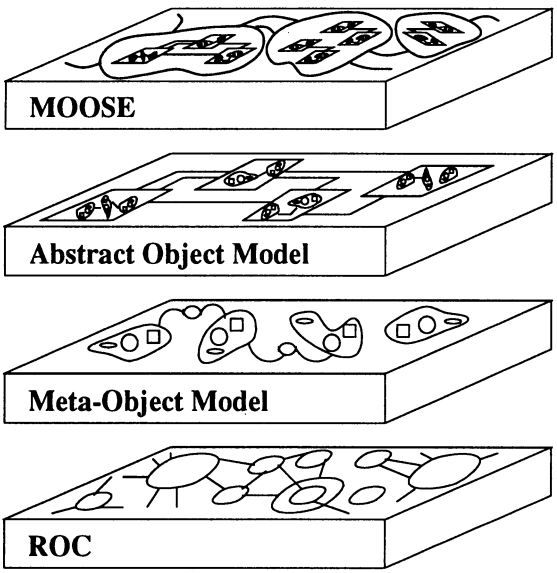

Figure 1 The layered semantics of MOOSE.

2 MOOSE OPERATIONAL FRAMEWORK

High assurance security requires a pervasive and practical application of formal methods to system execution models. The inherent heterogeneity of distributed systems makes this a daunting task. However, a unified view of distributed systems can simplify the task of achieving high assurance secure interoperation of distributed objects.

The operational framework of the Meta Object Operating System Environment (MOOSE) provides the needed unified view by blending formal operational semantics and denotational semantics in a hierarchical (layered) system architecture. The base of the architecture contains a primitive concurrency model with formal operational semantics. Each successive layer in the architecture is constructed with denotational semantics using the previous layer as a semantic algebra. This approach provides each layer of the framework – from the concurrency formalism up to the application level – with a formal operational semantics. Figure 1 illustrates the layered operational semantics of MOOSE.

The special requirements for the concurrency formalism underlying the MOOSE operational framework motivated the development of the Robust Object Calculus (ROC), a process calculus for objects. ROC is based on Nierstrasz's Object Calculus (OC) (Nierstrasz, 1991), and extends it by permitting robust encapsulation of agents, an essential feature of objects.

The ease with which ROC models complex message-passing distinguishes it from traditional process calculi (e.g., Milner's π-calculus (Milner, *et al.*, 1989). ROC's primary role is to pro-

vide primitive but formal semantics to concurrent object systems at all levels in the MOOSE operational framework.

Concurrent object models are endowed with ROC semantics through their denotational definitions. It is possible to directly give ROC semantics to any concurrent object-oriented programming language, but this may be too large a leap. Therefore, instead of applying ROC directly as a denotational semantics for concurrent programming languages, it is used to define a primitive ACTORS-like (Agha, 1986) meta-object model (MOM). MOM can be used to efficiently capture the semantics of existing concurrent object-oriented programming languages. In MOOSE, MOM functions as an abstract common substrate for the interoperation of heterogeneous distributed objects. It actually serves as the semantic algebra for the denotational semantics of each abstract object model resident in the MOOSE environment.

The next layer in the operational framework (currently under construction) contains the syntax and semantics for a sophisticated concurrent object-oriented programming language. The denotational definition of this language uses MOM as its semantic algebra. The language will comprise data and control primitives common to object-oriented programming languages as well as synchronization primitives for concurrent programming. It will also serve as a secure "mega-programming" language, containing constructs for the seamless interoperation of components with ROC or MOM semantics and housing a core set of security services and mechanisms. The richness of the language will make it suitable for the development of MOOSE distributed OS agents residing in the top level of the layered architecture. These agents will have even more complex behavior to ensure seamless and secure interoperation of heterogeneous applications in an open, distributed environment. The agents will be deployed as middleware resting between the operating system and software components.

3 ROBUST OBJECT CALCULUS

The Robust Object Calculus (ROC) is designed to supply a formal operational semantics to each layer in the operational framework. ROC advances existing process calculi, e.g., Milner's π-calculus (Milner, *et al.*, 1989) and Nierstrasz's Object Calculus (OC) (Nierstrasz, 1991), by supporting complex message-passing and a robust form of encapsulation for concurrently executing objects.

Robust agent encapsulation is critical to object-oriented systems. It is achieved by mandating that private services and values in objects be inaccessible to external objects, thereby ensuring that objects have well-defined interfaces. From the point of view of a process calculus, this requires a higher level of communication control between agents. The π-calculus achieves such communication control using restriction (unique-naming) to create globally unique labels for communication ports. However, since wildcard matching is not allowed, i.e., only identical ports may match, the capacity for complex message-passing in the π-calculus is severely limited.

ROC uses unbindable values to achieve robust agent encapsulation without sacrificing the ability to model complex message-passing. This feature advances OC's pattern-matching-based communication system. A ROC agent is encapsulated using restriction to create an unbindable globally unique identifier for the agent. Exposed values containing the unbindable identifier are guaranteed only to match patterns containing the globally unique identifier. Such a pattern can only exist outside the encapsulated agent if the agent has transmitted its identifier to an external

agent by making it bindable to an exposed name. Multiple identifiers can be used to achieve nested encapsulation which is essential to most object systems.

The following subsections describe ROC agent communication, syntax and inference rules for communication and reduction.

3.1 Agent Communication

Communication in ROC is based on pattern-matching. Complex message-passing is achieved by matching *values* to *patterns*.

Two agents may communicate when a value exposed by one matches a pattern exposed by the other. Values are nested tuples of *names*. Patterns are nested tuples of names and *wildcards*. The following notation is used: $a, b, c \in \mathcal{A}$ (agents); $m, n, p \in \mathcal{N}$ (names); $u, v \in \mathcal{V}$ (values); $x, y, z \in \mathcal{X}$ (patterns).

Definition A *value* is a name, an agent, a tuple of values, or a bindable value. A value does not contain wildcards at any level. The BNF definition of a value is:

$$v ::= n \mid a \mid [v_1, v_2, ..., v_j] \mid v\#.$$

Definition A *pattern* is a value, wildcard, or tuple of patterns. The BNF definition of a pattern is:

$$x ::= v \mid n? \mid [x_1, x_2, ..., x_i].$$

A wildcard, i.e., a placeholder for a value, is denoted by $n?$ ($n \in \mathcal{N}$). A *bindable value*, which can be bound to a placeholder, is denoted by $v\#$ ($v \in \mathcal{V}$). Bindable values are transmittable to agents with matching wildcard patterns. Values not tagged with "#" are called *unbindable values* because they cannot be matched by wildcards.

Matching of patterns and values is accomplished by the "$\sim$" operator. The semantics of the bindable symbol "#" are clarified in the matching rules below. Unbindable values can match bindable values, but cannot match wildcards.

Definition The *match* operator is denoted by "$\sim$". The matching rules are:

$$\begin{array}{lcl@{\qquad\qquad}lcl} v\# & \sim & v\# & v & \sim & v \\ v\# & \sim & v & v & \sim & v\# \\ v\# & \sim & n? & \vec{v}(\#) & \sim & \vec{x} \Leftrightarrow \forall i, v_i \sim x_i. \end{array}$$

The last matching rule applies to a tuple of values where the tuple is either bindable or unbindable. This condition is denoted by "(#)".

Pattern-matching is achieved by applying the rules given above. For example, $[m, [n]] \sim [m, [n]]$. On the other hand, $[m, n] \not\sim [m, [n]]$.

Wildcards can match bindable values. For example, $[m, [n]] \sim [m, [n]\#]$ and $[m, p?] \sim [m, [n]\#]$. On the other hand, $[m, p?] \not\sim [m, [n]]$.

Free occurrences of the wildcard are replaced with the value inside the accepting agent. When

$v \sim x$, the notation $a\{v/x\}$ is used to denote that all wildcards in x are replaced by the corresponding subvalues of v in the agent a. For example, when $v\#$ binds to $n?$ in a, each free occurrence of n in a is replaced by v.

3.2 Syntax

The ROC syntax is derived from Nierstrasz's Object Calculus (OC) (Nierstrasz, 1991). The symbol "#" distinguishes bindable values from unbindable values. The non-deterministic choice operator "+" from π-calculus is added. The BNF syntax is shown below.

$a ::= a \;\&\; a$	(concurrent composition)
$\mid n := a$	(recursion)
$\mid a + a$	(non-deterministic choice)
$\mid a \mid a$	(left preferential choice)
$\mid x \rightarrow a$	(input)
$\mid v\hat{}a$	(output)
$\mid a@v$	(application)
$\mid n\backslash a$	(new name n in a)
$\mid n$	(name)
$\mid \mathbf{nil}$	(empty agent)

The following notation is used in the syntax definition: $a, b, c \in \mathcal{A}$ (agents); $n, m \in \mathcal{N}$ (names); $v \in \mathcal{V}$ (values); $x, y, z \in \mathcal{X}$ (patterns). Note that the set of values is a proper subset of the set of patterns, i.e., $\mathcal{V} \subset \mathcal{X}$. The operators "&", ":=", "|", "+", "$\rightarrow$", "^" and "\" are right-associative. This order indicates the binding precedence from loosest to tightest. The application operator, "@", is left-associative. Its binding precedence is tighter than "^" and looser than "\". Agent communication may occur when an input pattern *matches* an output value.

The structural congruence rules shown below are used for manipulating expressions. They do not represent system activity, but are used to transform stable expressions into reducible states. Note that fn denotes a free name.

1. $a \;\&\; b \equiv b \;\&\; a, \quad a \;\&\; (b \;\&\; c) \equiv (a \;\&\; b) \;\&\; c$
2. $a + b \equiv b + a, \quad a + (b + c) \equiv (a + b) + c$
3. $n := a \equiv a\{(n := a)/n?\}$
4. $n\backslash a \equiv a, n \notin fn(a)$
5. $n\backslash m\backslash a \equiv m\backslash n\backslash a$
6. $n\backslash a \star b \equiv n\backslash(a \star b), n \notin fn(b), \quad a \mid n\backslash b \equiv n\backslash(a \mid b), n \notin fn(a)$
   where $\star \in \{\&, \mid, +, @\}$
7. $a \;\&\; \mathbf{nil} \equiv a, \quad \mathbf{nil}@v \equiv \mathbf{nil}$
8. $n := a \equiv n' := a\{n'/n\}, n' \notin fn(a)$
9. $n\backslash a \equiv n'\backslash a\{n'/n\}, n' \notin fn(a)$
10. $x \rightarrow a \equiv x\{n'/n\} \rightarrow a\{n'/n\}, n' \notin fn(x, a)$

Rules 1 and 2 address the commutativity and associativity of parallel composition and choice, respectively. Rule 3 is for expanding recursive agents. Rule 4 stipulates when a restriction can be discarded ($fn(a)$ denotes the set of free names in a). Rule 5 formalizes the commutativity of

restriction. Rule 6 describes scope extrusion. Scope extrusion expands the scope of a restriction to proximal agents. Rule 7 shows the effect of **nil** and any value applied to **nil**. Rules 8-10 define α-conversion for agents which substitutes globally unique names for local names. They are useful when scope extrusion is necessary.

3.3 Inference Rules

Inference rules for communication and reduction supply the semantics for agent expressions. Actions in the ROC universe are comprised of communication, reduction and binding. The rules are similar to those in OC except for an additional rule to handle non-deterministic choice.

Definition *Communication offers* are written as $\xrightarrow{\alpha}$, where α is either v (for input) or $\bar{v}$ (for output). *Reduction* is written as $\longrightarrow$. Communication offers and reduction are defined by the following rules:

In : $\dfrac{v \sim x}{x \rightarrow a \xrightarrow{v} a\{v/x\}}$ **Out** : $\dfrac{}{v\hat{}a \xrightarrow{\bar{v}} a}$

Conc : $\dfrac{a \xrightarrow{\alpha} a'}{a \;\&\; b \xrightarrow{\alpha} a' \;\&\; b}$ **Choice** : $\dfrac{a \xrightarrow{\alpha} a'}{a + b \xrightarrow{\alpha} a'}$

If : $\dfrac{a \xrightarrow{\alpha} a'}{a \mid b \xrightarrow{\alpha} a'}$ **Else** : $\dfrac{a \not\longrightarrow, a \not\xrightarrow{\alpha}, b \xrightarrow{\alpha} b'}{a \mid b \xrightarrow{\alpha} b'}$

Comm : $\dfrac{a \xrightarrow{v} a', b \xrightarrow{\bar{v}} b'}{a \;\&\; b \longrightarrow a' \;\&\; b'}$ **Apply** : $\dfrac{a \xrightarrow{v} a'}{a@v \longrightarrow a'}$

Left : $\dfrac{a \longrightarrow a'}{a \star b \longrightarrow a' \star b}$, $\star \in \{\&, |, +, @\}$ **Right** : $\dfrac{b \longrightarrow b'}{a \star b \longrightarrow a \star b'}$, $\star \in \{\&, |, +, \backslash\}$

Struct : $\dfrac{a \equiv b, b \longrightarrow b', b' \equiv a'}{a \longrightarrow a'}$

Conc provides concurrency by allowing "composed" agents to reduce independently. **Choice** allows only one of several possible activities. The **If** and **Else** rules prefer the reduction of the left side over the right side whenever possible. This is important for handling "default" actions (which is lacking in the π-calculus). **Comm** (global communication) matches complementary communication offers and reduces the system by realizing the communication. **Comm** works in concert with the **In** and **Out** rules. **Apply** (local communication) requires agent a to eventually accept value v. It cannot communicate externally using **Comm** until value v is consumed. If v is never consumed, then agent $a@v$ is effectively dead. The **Left** and **Right** rules allow for activity

within (or under) the various operators. The **Struct** rule allows an expression to be manipulated using the structural congruence rules so that further activity can occur when a "stable" agent is structurally congruent to a reducible one. The **Struct** rule manifests symmetrical inference rules for **Conc** and **Choice** where $a \;\&\; b \equiv b \;\&\; a$ and $a + b \equiv b + a$.

4 META-OBJECT MODEL

This section describes the Meta-Object Model (MOM), an ACTORS-like system (Agha, 1986) developed for the MOOSE operational framework. The model is defined as a system of ROC agents and, therefore, also serves to illustrate the construction of abstract models with ROC. The underlying principles of ROC, e.g., encapsulation and tuple-based communication, have facilitated the formal design of MOM. Using another process calculus to design MOM is much more difficult, if not impossible.

MOM is designed to serve as a primitive object architecture for constructing more sophisticated object models and programming languages. MOM is similar to the ACTORS system (Agha, 1986) and Chien's Concurrent Aggregates (Chien, 1993). It supports core object functionality, including persistence, method invocation, asynchronous message-passing, delegation and aggregation. Virtually any object system can be modeled with this core functionality. Thus, MOM is particularly suited to addressing interoperability issues.

4.1 Syntax and Notational Conventions

The ROC definitions of MOM agents use a special syntax for parameterized agents.Agents are parameterized with a standard functional interface.

$$agent(n_1, n_2, ..., n_m) := agent' \stackrel{\text{def}}{=} agent := n_1? \rightarrow n_2? \rightarrow ... \rightarrow n_m? \rightarrow agent'$$

The access of a parameterized agent is defined as:

$$agent(v_1, v_2, ..., v_m) \stackrel{\text{def}}{=} ((...((agent@v_1)@v_2)@...)@v_m)$$

Note that **keywords** are in bold type and VARIABLES are capitalized.

4.2 MOM Objects

MOM objects are viewed as a collection of tightly encapsulated agents. Each MOM object has a set of identifiers that defines how it can be addressed. Identifiers in MOM are navigational tuples of names (this bears the influence of tuple-based communication in ROC). The syntax of navigational identifiers (nids) in MOM is defined by the following BNF rule:

$$nid ::= [\ [\mathbf{out}, \mathbf{root\#}], \mathbf{nil\#}] \mid [\ [\mathbf{out}, \mathbf{def\_par}], \mathbf{nid\#}] \mid [\ [\mathbf{out}, \mathbf{lid\#}], \mathbf{nid\#}] \mid [\ [\mathbf{in}, \mathbf{lid\#}], \mathbf{nid\#}] \mid \mathbf{nil\#}$$

Every MOM object, except for the root object, resides inside some other object. Each object is given a local identifier (lid) unique to its domain. This forms the basis for a system of domains.

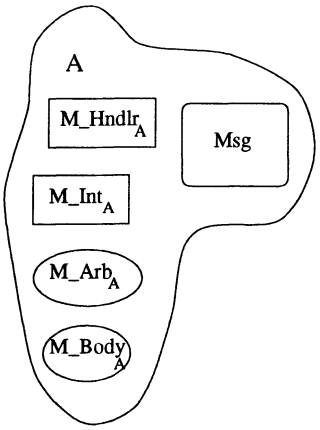

Figure 2 MOM object components.

The domain of an object is characterized by its parental identifier. The keyword **def_par** is used by the definitions as a pronoun for an object's parent and the keyword **root** is used as a pronoun for the root object.

A MOM object is given an atomic name to serve as its local identifier (lid), e.g., Obj_1. This name is prepended to its parental identifier, e.g. [[**out**, Obj_1#],[[**out**, **root**#], **nil**#]#] to create a unique global identifier for the object. All MOM systems have only one root object. Along with identifiers, an object must contain *method bodies*, *method arbiters*, method interfaces and *message handlers*. *Messages* also reside in objects, but their existence is more transitory. The core MOM components are shown in Figure 2.

4.3 MOM Messages

A MOM message is a persistent entity in the system. All messages exist until they are consumed by a message handler, i.e., all communication is *asynchronous*. Messages are categorized as requests, replies or acknowledgements. They are defined using a composition of subagents and patterns following modular design principles. The ROC definition of MOM messages is given below. The message content definition, i.e. the definition for Msg_Body, is omitted to simplify the presentation.

$$Msg(lid_{Msg}, pid, src, dest, c\_body) := [\ [lid_{Msg}\#, pid], [\mathbf{msg}, \mathbf{msg\_h}], [src\#, dest\#, Msg\_Body\#]\#]\hat{}\mathbf{nil}$$

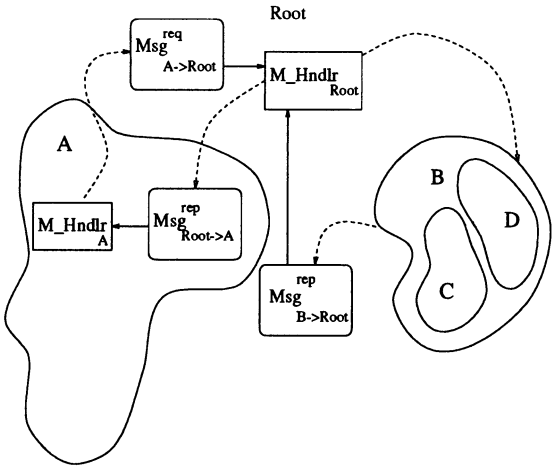

Figure 3 Messaging in MOM.

4.4 Message Handlers

A message handler processes incoming messages and marshalls object requests. It forms the basis for an object's identity as it controls the distribution of requests and replies for a tightly bound set of agents. The creation and acceptance of messages by a message handler (M_Hndlr_A) is illustrated in Figure 3.

An incoming message can be turned into a method invocation or delegated to another object. By delegating a message, a message handler consumes the old message and creates a new one in an adjacent domain. For example, the root domain's message handler M_Hndlr_{root} consumes a reply message $Msg^{rep}_{B \to root}$ and creates a new message $Msg^{rep}_{root \to A}$ in object A's domain.

Error handling is incorporated in MOM object message handlers: If the destination of the message does not exist, the originating message handler must accept the message. A MOM object's methods can issue a request to a message handler which must then be turned into a message for delivery. The message handler is designed as a complex ROC agent, comprising many subagents performing specific tasks. The definition of the MOM message handler is given below.

$$
\begin{aligned}
Msg\_handler(pid) \;:=\; & ([[\, LID_{Msg}?, pid], [\mathbf{msg}, \mathbf{msg\_h}], [SRC?, \mathbf{nil}, BODY?] \,] \\
& \rightarrow \; (Rcv\_Msg@BODY\# \;\&\; Msg\_handler(pid))) \\
& \mid \; ([[\, LID_{Msg}?, pid], [\mathbf{msg}, \mathbf{msg\_h}], CONTENT?] \\
& \rightarrow \; (Dlg\_Msg@CONTENT\# \;\&\; Msg\_handler(pid)))
\end{aligned}
$$

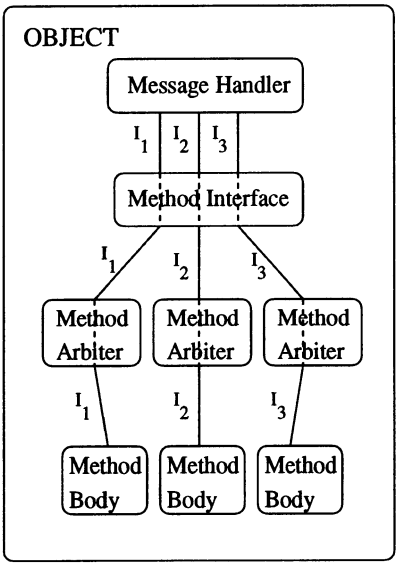

Figure 4 MOM method scheme

In the definition above, the ROC agent *Dlg_Msg* sends a copy of the message to the next domain in the message's path in the event that it has not reached its final destination. The agent *Rcv_Msg* handles the situation where a method invocation is requested by exposing a method invocation request pattern. The definitions of *Dlg_Msg* and *Rcv_Msg* are omitted for brevity.

4.5 Method Agents

A method invocation request created by a message handler is received by a method interface (see Figure 4). The method interface can be used to synchronize method access. It creates a method arbiter when a method invocation occurs. The method arbiter handles reply, request and acknowledgement communications for individual method invocations.

A MOM method performs operations on primitive data types (e.g., integers, strings, arrays, etc.) and/or issues other method invocation requests. The method interface is responsible for accepting invocation communications from the message handler which spawn method arbiters, one for each invocation. The method arbiter then spawns a method body. The method interface is used for controlling access to individual methods. A unique method interface exists for each method in an object.

Each method invocation spawns a new method body and arbiter. However, the method interface is a singular persistent entity. The completion of an invocation results in a reply from the method to the method arbiter. This reply can be propagated back to the initiating object to implement function calls. Agents *Method_Arbiter* and *Method_Interface* are defined below. Subagent definitions are omitted to simplify the presentation.

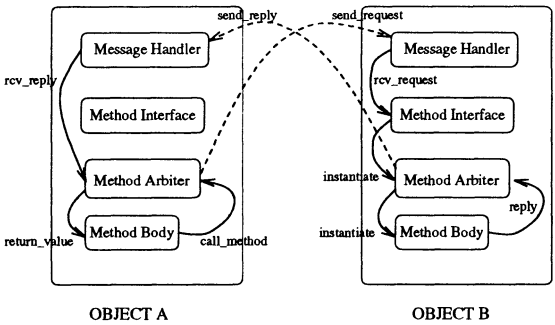

Figure 5 Method invocation.

$$
\begin{aligned}
Method\_Arbiter \quad & (lid_{MA}, pid) := \\
& [LID_{MH}, P1?, P2?, ..., REPLYTO?, ACKTO?] \rightarrow \\
& (Send\_Ack@[ACKTO\#] \\
& \ \& \ lid_{MB} \backslash \ Method\_Body(lid_{MB}, pid)@[\ lid_{MA}\#, P1\#, P2\#, ...] \\
& \ \& \ (MA\_Return + MA\_Call)
\end{aligned}
$$

$$
\begin{aligned}
& Method\_Interface(lid_{MI}, pid, Method\_Body, Method\_Name, Type\_1, Type\_2, ...) := \\
& (\ [\ [LID_{MH}?, pid], [\mathbf{msg\_h}, \mathbf{MI}], \\
& \ [method\_name, [\ [type\_1, P1?], [type\_2, P2?], ...]\ REPLYTO?, ACKTO?]\ ] \rightarrow \\
& (Method\_Interface(lid_{MI}, pid, Method\_Body, Method\_Name, Type\_1, Type\_2, ...) \\
& \ \& \ lid_{MA} \backslash Method\_Arbiter(lid_{MA}, pid)@[\ LID_{MH}\#, P1\#, P2\#, ..., \\
& REPLYTO\#, ACKTO\#]
\end{aligned}
$$

Method bodies can also be modeled by ROC. The only restriction is that they must strictly conform with the communication interface specified by the method interface, i.e., they must be properly encapsulated. A method body can request a service from a foreign object, i.e., it can invoke a method in a foreign object. Method invocation is illustrated in Figure 5.

To return from an invocation, a method must return a value, which it does by means of *return_value*. A method body can invoke another method with *call_method*. The method body must create a communication channel for a method call by offering *channel**call_method*(*channel*, *gid*,*method_name*,*params*). The method body then awaits a reply, exposing a pattern of the form [*channel*, *RETURN_VAL*?] for input into the method body.

5 DISTRIBUTED OBJECT SECURITY

This section clarifies how the layered MOOSE architecture is used to achieve high assurance security in heterogeneous distributed systems. The focus is on secure interoperability of distributed objects. The following subsections summarize the main issues pertaining to secure interoperability and describe the MOOSE verification framework for creating high assurance secure interoperation of distributed objects.

5.1 Secure Interoperability

Seamless and secure interoperation will be the underlying theme of future computer systems. Many Internetworked systems have some degree of transparent interoperation, but few, if any, can seamlessly and securely interoperate with each other. For two systems (or components) to interoperate, each must be capable of sending messages that the other can understand and process. Secure interoperation is the cornerstone of distributed object security. It mandates that systems send and understand messages without the potential for violating other systems' security policies. These security policies may be specified using natural language or, better yet, formal semantics.

New problems arise when components are distributed amongst heterogeneous systems. Each system might have a different security policy and/or model. When this happens, secure interoperation requires *policy negotiation*. The negotiation process establishes a consensus security policy for interoperation from the competing policies. Obviously, the negotiated policy must satisfy the general security requirements of each of the involved parties.

Policy negotiation can be quite complicated when models are non-comparable. The difficulty lies in achieving a common semantic basis for subjects and objects. Consider, for example, the problem of reconciling two arbitrary sensitive information labeling schemes. Even if a common labeling scheme exists, no guarantees can be made that what is secret in one system should also be secret in the other system. This example only addresses access control. To achieve secure interoperation, each security service offered by interoperable components must be negotiated to obtain a consensus.

The concern for distributed object interoperability has produced various management schemes, including OMG's CORBA (Object Management Group, 1991; Mowbray and Zahavi, 1995) and OSF's DCE (Open Systems Foundation, 1992; Rosenberry, Kenney and Fisher, 1993). These schemes allow architecture-compliant objects and systems to interoperate seamlessly. However, the impetus to provide practical solutions has caused security to take a back seat to interoperability. Furthermore, security models and mechanisms for these architectures are hindered by a lack of formal semantics. This often results in vague, ambiguous specifications that only limit high assurance performance. Our plan to provide CORBA, DCE and other emerging architectures with a common formal foundation in MOOSE is an important first step to achieving seamless and secure interoperation in heterogeneous distributed object systems.

5.2 Verification Framework

The operational framework of MOOSE, consisting of execution model layers with formal operational semantics, has a companion verification framework used for reasoning about the system (Figure 6). The foundation of the operational framework is the ROC process calculus for concur-

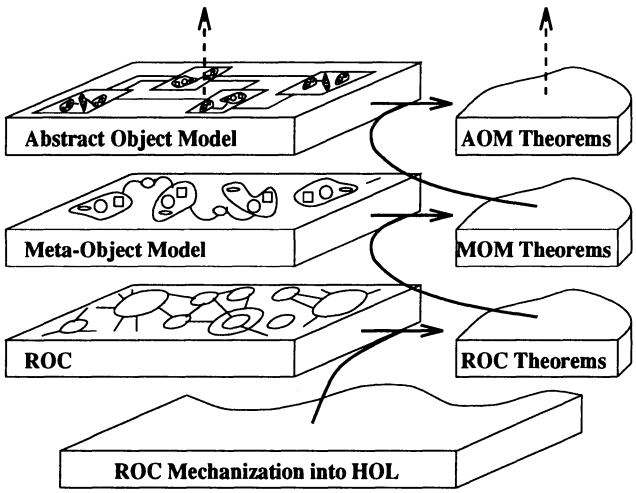

Figure 6 HOL semantic hierarchy and operational framework.

rent objects. Each successive layer is given a denotational semantics using the immediately lower layer as a semantic algebra. This scheme gives formal operational semantics to systems in each layer.

Using a formal operational semantics alone limits the potential for system specification and verification. Certain system properties, most notably simulation and bisimulation properties, can be proved with formal operational semantics. Still, operational semantics are constrained because they must be executable. Axiomatic semantics (logics) do not suffer from this limitation. They are more expressive and better suited to specification and verification.

ROC semantics is easily axiomatized to permit reasoning by an interactive HOL theorem prover (Melham, 1992). The HOL system provides a suitable environment for reasoning about computational systems (Gordon and Melham, 1993). It uses a higher order logic based on Church's logic of types (Church, 1940) which extends predicate logic by adding types and letting variables range over functions.

The axiomatization of ROC semantics into higher order logic is achieved by a mechanization processs. A purely definitional approach ("deep embedding") is adopted in which the ROC agent system is expressed as a defined type within the logic. The principal benefit of this approach is that no new axioms need to be introduced into the logic. Only axioms that come directly from ROC are used.

HOL semantics for ROC brings us one step closer to distributed application reasoning. Reasoning about these complex systems at the ROC level, however, is not practical. Therefore, a hierarchical system of HOL semantics is used for reasoning about distributed applications. This system mirrors the layered operational framework.

As with the operational framework, the layers in the verification framework begin with the foundational ROC process calculus. The bottom layer contains the HOL semantics created by the mechanization of ROC. It also contains derived semantics about ROC and systems built from

it. The next layer associated with the Meta-Object Model (MOM) contains theorems about MOM and MOM systems. These theorems are constructed using the denotational semantics of MOM and the HOL theorems in the previous layer.

Obviously, the kinds of theorems contained in each layer will affect the ability to derive more theorems at higher levels. As more abstract object models tend to emerge from compositions of objects, it will be necessary to focus on theorems that respect object composition.

The relationship of the HOL semantic hierarchy with the operational framework is illustrated in Figure 6. Practical application verification can be accomplished only by reasoning at higher levels. The HOL semantic hierarchy used in MOOSE is a bootstrapping approach that is particularly suited to application-level reasoning.

5.3 High Assurance Security

High assurance security must consider security in all its forms: *security mechanisms*, *functions*, *services*, *models* and *policies*. A security mechanism implements security functionality that might be offered as part of some security service, e.g., encryption as part of a secure communication service. A configuration of security services implements a security model which adheres to a security policy. A security policy specifies rules pertaining to the use and availability of sensitive information. Security flaws could exist anywhere in this infrastructure.

The key to achieving high assurance security for distributed objects is the pervasive application of formal methods. The main problem facing secure interoperation is the lack of a common semantic foundation for heterogeneous systems. Formal methods provide a foundation that facilitates transparent and reliably secure interoperation. The verification framework described above is a viable methodology for the pervasive application of formal methods to system verification.

The two parts to high assurance security are "design verification," i.e., proving that a system's formal specification satisfies its security policy's formal specification, and "implementation verification," i.e., proving that a system's implementation satisfies its formal specification. Defining security mechanisms within the operational framework makes implementation verification possible. Services constructed from security mechanisms may be endowed with formal semantics, also permitting implementation verification.

At higher levels, security models may be given abstract ROC semantics while security policies may be expressed using HOL. This permits design verification of security models. With this combination of design and implementation verification, a distributed system can be shown to satisfy a formal security policy.

Figure 7 illustrates the security verification methodology. Note that the security services offered by DCE and CORBA object request brokers (ORBs) occupy a semantic layer above the Meta-Object Model (MOM). Theorems can be derived about these security services and the object models that use them. The new theorems allow the derivation of additional security-related theorems for distributed applications which can be used to verify key properties of system security policies.

Two components that are in systems employing different security policies, but that adopt the rigorous verification methodology above, are candidates for computer-assisted policy negotiation. The framework assists this process by providing a common semantics in which disparate models can reconcile clearance and sensitivity equivalences. Furthermore, there is now a formal basis for comparing security services, so that a negotiating system can specify all the services that a sensitive object will require.

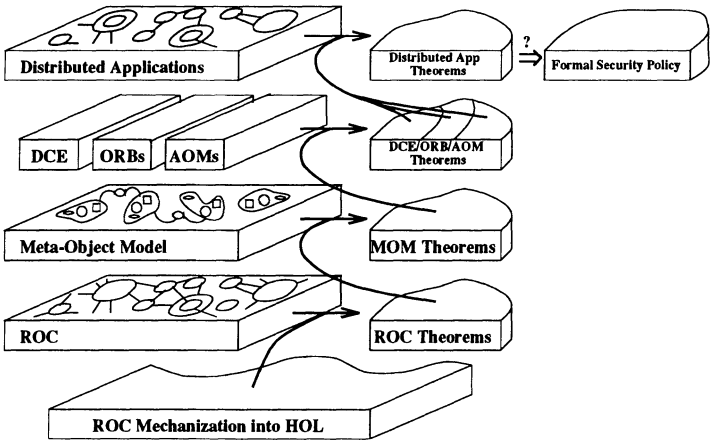

Figure 7. Security verification methodology.

6 COMPARISON WITH OTHER APPROACHES

Interoperability is not a new concept, although the advent of object technology has drastically changed the computer system landscape. The clean interfaces provided by objects have raised expectations, and rightfully so, that heterogeneous software components should interoperate securely. Distributed object architectures address secure interoperability, but in an informal way that can pose hazards when dealing with other architectures.

Work in verifiably secure distributed systems has provided powerful methodologies for highly integrated system verification, but these methodologies have yet to be applied to the secure interoperation of distributed objects. This section summarizes the major efforts in the areas of distributed object interoperation and verifiably secure distributed systems, and relates them to on-going work in the MOOSE project.

6.1 Distributed Object Architectures

The need for secure interoperation has produced various "standard" architectures, the most prominent being OMG's CORBA (Open Systems Foundation, 1992; Rosenberry, Kenney and Fisher, 1993) and OSF's DCE (Open Systems Foundation, 1992; Rosenberry, Kenney and Fisher, 1993). These schemes allow compliant objects and systems to interoperate. Security is considered in CORBA and DCE, but is applied in a somewhat *ad hoc* manner.

CORBA provides a standard architecture for distributed object interaction (Open Systems Foundation, 1992; Rosenberry, Kenney and Fisher, 1993). However, it is not a panacea for secure distributed object interoperation. While it offers a security policy specification, CORBA lacks a formal semantics. The Distributed Computing Environment (DCE) addresses interoperability using middleware that provides a common environment for heterogeneous computer systems

(Open Systems Foundation, 1992; Rosenberry, Kenney and Fisher, 1993). DCE provides security mechanisms, but an inadequate formal foundation.

Newer object architectures which are continuing to emerge only make object interoperation more complicated. In fact, they create the new problem of securely interoperable standards. These schemes offer security mechanisms, models, and policies, most of them without any formal foundation and certainly no common foundation. There can be no guarantee that heterogeneous distributed object management schemes will interoperate securely (if at all) without a common formal foundation.

CORBA and DCE will continue to evolve (and co-exist), and new distributed object management schemes will proliferate. Only a handful of these schemes will likely dominate, but the heterogeneity and *ad hoc* nature of the underlying models and standards will make seamless and secure interoperability virtually impossible. The common formal foundation provided by MOOSE sets the stage for a more general notion of secure interoperability in heterogeneous distributed object systems.

6.2 Verifiably Secure Distributed Systems

Work in verifiably secure distributed systems applies formal methods to abstract models of distributed systems. This is usually done by modeling security services and execution models of distributed systems with formal semantics and providing formal security policy specifications. If the formal semantics of the service satisfies the policy specification, then the system is proven to be secure.

The Silo project (Zhang, *et al.*, 1994, 1995) presents a useful hierarchical verification methodology for distributed systems based on formal methods (Bevier, *et al.*, 1989; Alves-Foss and Levitt, 1991). The methodology prescribes a semantic layer for each computational substrate, from the hardware level up to the application level. Each layer can be formally specified as an abstract machine defined from the layer beneath it. MOOSE advances the Silo effort by using a process calculus tailored to distributed and concurrent objects as the foundation for its operational framework.

Theoretical work on verifiably secure distributed systems explores formalisms for computer security and new methods for reasoning about distributed systems. Researchers at NRL's Center for High Assurance Computer Systems (Maclean and Meadows, 1989; Maclean, 1990) have proposed practical methods for formal security model specification based on the formal foundations of computer security. In particular, they promote compositional reasoning as a critical technology for efficient distributed system verification. Related research by Gong and Qian at SRI (Gong and Qian, 1996) has focused on the theoretical implications of secure interoperability between heterogeneous systems. This work also espouses the principle of compositional reasoning for distributed systems and examines the complexity of various reasoning techniques applied to heterogeneous distributed systems.

These research efforts and others in the area of verifiably secure distributed systems have developed important techniques for achieving high assurance distributed system security. The MOOSE project applies them to the domain of distributed objects.

7 CONCLUSIONS

The operational and verification frameworks of MOOSE provide a powerful methodology for developing heterogeneous distributed object systems with high assurance secure interoperability. The main features of this work are its use of a hierarchical proof system and ROC, a process calculus tailored to concurrent objects which gives each semantic layer in the operational framework a formal foundation. The verification framework contains higher order logic (HOL) semantics for each operational semantic layer. Axiomatic semantics for ROC are derived through a mechanization of ROC into HOL.

The frameworks are readily applied to high assurance secure interoperability by modeling various security mechanisms, services, models and policies within the framework. Security policies are given HOL semantics, while HOL semantics for security model implementations are derived from the frameworks. This approach facilitates the verification of security policy implementations and policy negotiations.

Current work on the MOOSE project involves constructing the upper layers of the frameworks. For the operational framework, this entails modeling various programming languages, security mechanisms and services and popular distributed object architectures, e.g., CORBA and DCE. An implementation for the operational framework to run on heterogeneous UNIX platforms is planned. This will rely on a virtual machine (ROCVM) that efficiently executes ROC expressions.

Secure interoperation between heterogeneous distributed objects is critical to current and future computer systems, mandating high assurance performance. Formal methods provide the technology for high assurance computing. The MOOSE framework allows the practical application of formal methods to high assurance computing in heterogeneous distributed systems.

Acknowledgement This research was supported by OCAST Grant AR2-002 and MPO Grants MDA904-94-C-6117 and MDA904-96-1-0115.

REFERENCES

Agha, G.A. (1986) *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts.

Alves-Foss, J. and Levitt, K. (1991) Mechanical verification of secure distributed systems in higher order logic. *Proceedings of the 1991 International Workshop on Higher Order Theorem Proving and its Applications*, 263–278.

Bevier, W.R., Hunt, W.A., Moore, J.S. and Young, W.D. (1989) An approach to systems verification. Technical Report 41, Computational Logic, Inc., Austin, Texas.

Chien, A.A. (1993) *Concurrent Aggregates*. MIT Press, Cambridge, Massachusetts.

Church, A. (1940) A formulation of the simple theory of types. *Journal of Symbolic Logic*, **5**, 56–68.

Diller, A. (1990) *Z: An Introduction to Formal Methods*. John Wiley, New York.

Gong, L. and Qian, X. (1996) Computational issues in secure interoperation. *IEEE Transactions on Software Engineering*, **32**, 43–52.

Gordon, M. and Melham, T.F. (eds.) (1993) *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, Cambridge, U.K.

Hinchey, M. and Bowen, J. (eds.) (1995) *Applications of Formal Methods*. Prentice Hall, New York.

Maclean, J. (1990) The specification and modeling of computer security. *IEEE Computer*, **23(1)**, 34–46.

Maclean, J. and Meadows, C. (1989) Composable security properties. *Proceedings of the Computer Security Workshop.*

Melham, T.F. (1992) A mechanized theory of the π-calculus in HOL. Technical Report 244, University of Cambridge Computer Laboratory, Cambridge, U.K.

Milner, R., Parrow, J. and Walker, D. (1989) A calculus of mobile processes. Technical Report ECS-LFCS-89-85&86, University of Edinburgh, Edinburgh, U.K.

Mowbray, T.J. and Zahavi, R. (1995) *The Essential CORBA: Systems Integration Using Distributed Objects*. John Wiley, New York.

Nierstrasz, O. (1991) Towards an object calculus, in *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing* (eds. M. Tokoro, O. Nierstrasz and R.A. Olsson), Springer Verlag, Amsterdam, The Netherlands, 1–20.

Object Management Group and X/Open (1991) The common object request broker: Architecture and specification. Technical Report OMG Document No. 91.12.1, Object Management Group and X/Open, Framingham, Massachusetts.

Open Systems Foundation (1992) The OSF distributed computing environment. Technical Report OSF-DCE-PD-1090-4, Open Systems Foundation, Cambridge, Massachusetts.

Rosenberry, W., Kenney, D. and Fisher, G. (1993) *Understanding DCE*. O'Reilly and Associates, Inc., Sebastopal, California.

Wing, J. (1990) A specifier's introduction to formal methods. *IEEE Computer*, **23(9)**, 8–21.

Zhang, C., Shaw, R., Heckman, M.R., Levitt, K. and Olsson, R.A. (1995) A hierarchical method for reasoning about distributed programming languages and applications. *Proceedings of the 1995 International Workshop on Higher Order Logic Theorem Proving and its Applications.*

Zhang, C., Shaw, R., Heckman, M.R., Benson, G.D., Archer, M., Levitt, K. and Olsson, R.A. (1994) Towards a formal verification of a secure distributed system and its applications. *Supplementary Proceedings of the Seventh International Workshop on Higher Order Logic Theorem Proving and its Applications.*

8 BIOGRAPHY

John Hale is a Ph.D. candidate in computer science at the University of Tulsa. His research interests are in database security, heterogeneous distributed systems and computer graphics. Mr. Hale received his B.S. (CS) and M.S. (CS) degrees from the University of Tulsa in 1990 and 1992, respectively.

Jody Threet is a Ph.D. candidate in computer science at the University of Tulsa. His research interests are in distributed systems, intelligent systems and formal methods. Mr. Threet, a founding partner of Sleek Software, Inc., Austin, Texas, received his B.S. (Math) and M.S. (CS) degrees from the University of Tulsa in 1990 and 1993, respectively.

Sujeet Shenoi is an associate professor of computer science at the University of Tulsa. He received his B.Tech. degree from the Indian Institute of Technology, Bombay in 1981, and his M.S. (Ch.E.), M.S. (CS) and Ph.D. degrees from Kansas State University in 1984, 1987 and 1989, respectively. Dr. Shenoi's primary research interests are in database systems, artificial intelligence and intelligent control.