Object-Oriented Extendibility in Hermes/ST, a Transactional Distributed Programming Environment *

Michael Fazzolare, Bernhard G. Humm and R. David Ranson

Telecommunications Software Research Centre (TSRC)
The University of Wollongong, Department of Computer Science
P.O. Box 1144 (Northfields Avenue), Wollongong NSW 2522, Australia
Tel: (+61 42) 26 8832, Fax: (+61 42) 27 3277
Email: {faz, humm, dranson}@cs.uow.edu.au

Abstract. A major aim of transactional distributed programming environments is to facilitate the development of reliable distributed applications by shielding the developer from concerns such as failures. This paper describes the linguistic features of the Hermes/ST object-oriented distributed programming environment that further ease the development of such applications by enhancing the flexibility and extendibility of their implementations. This is achieved through the parameterisation of properties such as permanence, concurrency, transactional semantics and distribution. Parameterisation supports reuse, and enables the notion of incremental development, whereby a simple centralized sequential prototype of the application can be easily validated before being gradually extended to the final efficient reliable distributed application. An example application is included to demonstrate this approach.

1 Introduction

Concurrent and distributed applications are inherently more complex than their centralized, sequential counterparts. This added complexity manifests itself particularly in the programming and debugging of such applications. A major aim of distributed programming environments is to reduce the difficulty of programming such applications by providing constructs that shield the programmer from concerns such as fault-tolerance and concurrency control. Transactions [GR93], for example, are a common construct that distributed programming environments provide to maintain the consistency of distributed data in the presence of concurrency and partial failure.

Hermes/ST is a distributed programming environment that supports an incremental development strategy as a means to ease the prototyping and implementation of distributed applications. The incremental development of a distributed application involves starting with a distributed design of the application,

* This research was funded by Telecom Australia under Contract No. 7260.
implementing a centralized sequential prototype of that design, and then gradually extending this prototype to the final distributed, concurrent fault-tolerant application. The functional correctness of the design and initial implementation can be determined far more easily in a non-concurrent, centralized environment. Once these have been validated, Hermes/ST supports the incremental addition of features such as permanence, concurrency, transactional semantics and distribution.

Hermes/ST's incremental development strategy is flexible. The movement from one development stage to the next does not involve structural changes to code. Rather, through parameterization, Hermes/ST allows the addition of extra features simply by changing parameters. Hermes/ST provides convenient general strategies, such as implicit locking, which satisfy a large class of required behaviours. It also complements these strategies with facilities for their easy extension. Thus, for example, implicit concurrency control can be extended by the addition of explicit concurrency control. At each stage in the development process, versions of the system can be debugged and validated, and if necessary optimized. Optimization can be achieved, for example, by the fine-tuning of subtransactional checkpointing, explicit concurrency control, transactional retries and timeouts.

Hermes/ST also enhances the reusability of classes in different contexts. An example of reuse in Hermes/ST is the ability of Hermes/ST classes to be instantiated as both persistent and volatile objects. A further example of reuse is the ability of Hermes/ST methods to be invoked both transactionally and non-transactionally. This is achieved through parameterization.

The next section describes an example distributed banking application that illustrates and clarifies many aspects of Hermes/ST outlined in this paper. The following sections describe the linguistic mechanisms and features of Hermes/ST that support these notions of incremental development and reusability. The Hermes/ST object model (Section 3), generalized invocation scheme (Section 4) and concurrency control specification (Section 5) are described. Significant code from our distributed banking example is included in the appendix. The features are compared to other distributed transactional systems, in particular to the Argus [Lis88], Avalon/C++ [EME91], and Arjuna [Shr92] distributed programming environments.

2 A Banking Example

In order to clarify various issues throughout the paper, an example application, an international bank, is regularly referred to. This application is often used as a test application for distributed programming environments [Lis88, EME91, Hew91]. The banking example described in this paper is derived from the banking system in [Lis88].

An electronic bank is composed of branches and teller machines, which are geographically distributed. Each branch and teller can communicate with any other. Each branch stores a collection of accounts. Accounts are identified by
their branch code and account name, and are either cheque or interest bearing savings accounts. Teller machines are used to open and close accounts, deposit, withdraw and (internationally) transfer money. A special teller, the main office, has knowledge about all branches in the bank, and provides special managerial functions such as conducting audits.

3 The Hermes/ST Object Model

The Hermes/ST object model has been inspired by the object model of Smalltalk-80 [GR89]. Indeed, Hermes/ST has been implemented as a set of Smalltalk classes that operate in the Smalltalk environment. However, the linguistic features of Hermes/ST described in this paper are independent of Smalltalk's features.

Hermes/ST objects contain methods, and encapsulate instance variables. Objects can have two kinds of instance variables: named variables and indexed variables, e.g., for array construction. Variables must be accessed through read and write access methods, e.g., self accountName (read access) and self accountName:newValue (write access) for named variables\(^2\), and at: and at:put: for indexed variables. Variables can only refer to other objects. This property has been called uniform reference semantics [Mey88].

Hermes/ST objects communicate via message passing, employing a standard or extended Smalltalk syntax. Hermes/ST's objects, variables and methods are defined in Hermes/ST classes. These classes are arranged in a single-inheritance hierarchy, and are descended from the Hermes/ST class HermesObject. Instance creation is achieved by calling a class method, which returns a reference to the newly created instance of that class. By using an instance creation parameter, Hermes/ST classes can be instantiated to return a reference to either a volatile or persistent instance. For example, the Hermes/ST statement BinTree instantiate:#volatile creates a volatile BinTree object and returns a reference to it whereas BinTree instantiate:#persistent creates a persistent BinTree object and returns a reference to it. Persistence or volatility of a Hermes/ST object is henceforth referred to as its kind.

Hermes/ST persistent objects have a back-up version on disk that closely follows their in-memory representation. This allows them to be longer-lived than their creating processes. They have state restoration handlers, which allow them to survive node or transaction failures. Because of their permanence, they must be deleted explicitly, rather than garbage collected as is the case for volatile objects. They exhibit location transparency [CC91], being accessible remotely, and are also remotely instantiable. Each persistent object possesses a network-wide unique persistent object reference. This reference has a symbolic representation.

\(^2\) Smalltalk programmers may wish to note that specific access methods are provided for all Hermes/ST classes via a set-up routine. Redefining the semantics of the assignment and instance variable read access would have provided cleaner syntax. However, this would have required modifying the Smalltalk compiler which is beyond the scope of this work.
Clients on remote nodes can access an object provided they know its symbolic reference and its public interface. Hermes/ST persistent objects are concurrency controlled, and support transactions.

Transactions provide serializability, atomicity and permanence properties. Serializability ensures that concurrent transactions appear to have run in some serial order. Atomicity ensures that either all or none of a transaction's actions are performed. Permanence ensures that once a transaction has completed, none of its effects will be lost due to non-catastrophic failure. Together, these properties allow a system to move from one consistent state [Mos85] to another. Transactions are specified as parameters to Hermes/ST method invocations (see Section 4). In addition to single level transactions, Hermes/ST persistent objects support nested transactions [Ree78, Mos85].

Hermes/ST volatile objects are similar to traditional Smalltalk objects. They are only accessible locally, are not concurrency controlled, have no state restoration, and do not support transactions. They are not saved on disk and are automatically garbage collected by the system.

3.1 Support for Incremental Development

The Hermes/ST object model makes the system particularly well suited to incremental development. This approach was used successfully in the implementation of the distributed bank described in this paper, and is being employed in several other ongoing experiments. After completing the design of the distributed application, a single-machine sequential prototype of the application is first implemented using volatile objects. This is debugged, and the design is at least partially validated. Detection and removal of design and implementation errors, many of which are not directly related to the distributed, concurrent or fault-tolerant nature of the application, is performed. The debugging/design validation process at this stage is greatly eased because it is performed on a single machine without concurrency, distribution and fault tolerance.

This validated prototype may then be extended. Implicit concurrency control and permanence are added by changing instantiation parameters from #volatile to #persistent. Structural changes to the code, and the bugs that these tend to introduce, are avoided through Hermes/ST's parameterised instantiation approach. After testing of this new prototype, distribution can be added likewise, or explicit concurrency and fault tolerance properties can be added to the application (see Sections 4 and 5).

3.2 Reuse Advantages

Some classes, such as collections, are suitable for use as both persistent and volatile objects. Hermes/ST's parameterised instantiation allows the specification of such classes. The BinTree and BinTreeNode classes used in our bank application demonstrate this support for reuse. See Appendix A.1 for the definition of some instance methods for BinTree.
The classes BinTree and BinTreeNode together define a persistent or volatile sorted binary tree. A BinTree instance contains the root of the tree. BinTreeNode instances form the nodes of the tree, which contain the elements of the collection, as well as references to left and right subtrees (BinTreeNode).

The instance creation BinTree class method, instantiate:withContents: creates a new binary tree instance of kind kind (either #persistent or #volatile) to contain anObject. It assigns a reference to a new binary tree node instance of the same kind to its variable root. The add:ifExisting: instance method similarly checks its own kind before creating a new node of the same kind. The remove:ifAbsent: instance method contains explicit invocations of method delete. delete does nothing in the case of a volatile object (which will be automatically garbage collected by the system), or removes the version of the object on disk in case of a persistent object.

Reusability is achieved through the parameterization of instance creation methods, and through the use of explicit deletion messages. Thus, a volatile binary tree is returned if #volatile is passed to BinTree's instance creation method. Alternatively, passing #persistent to this method returns a permanent concurrency-controlled binary tree that manages highly concurrent transactional accesses. Furthermore, traditional object-oriented support for reuse through inheritance is not compromised (e.g. subclasses of BinTree can be instantiated as persistent or volatile by inheriting instance creation methods).

3.3 Comparison to Other Approaches

Argus, Avalon/C++ and Arjuna do not support class or type specifications that are reusable for various “kinds” of objects. Argus is object-based, and thus does not support inheritance. Argus has object-based guardians, and data types within these guardians. It possesses atomic and volatile data types, but “kind” is a type property. Avalon/C++ is similar to Argus, with servers comparable to Argus’ guardians. “Kind” in Avalon/C++ is a static class property, with persistence properties defined through inheritance from recoverable, atomic or subatomic base classes. Arjuna is based on C++, and currently supports single inheritance. The “kind” of a class is also defined through its inheritance hierarchy. Arjuna supports volatile C++ classes, as well as persistent classes descended from LockManager.

The introduction of multiple inheritance could conceivably allow Avalon/C++ and Arjuna to reuse a class with varying “kinds” of instances. For example, a class VolBinTree could inherit from both classes BinTree and VolatileObject, while a class PersistentBinTree could inherit from classes BinTree and PersistentObject. However, when the object model is extended to include a large number of “kinds”, this approach would become cumbersome. It would require a large number of new subclasses — one for every “kind”.

3 If the deletion of a persistent object is requested during a transaction, it is delayed until the top-level transaction commits.

4 An extension of the Hermes/ST object model, currently being developed, extends the range of object kinds from the current extremes of “light-weight” volatile objects and “heavy-weight” persistent objects to a larger number of kinds.
4 A Generalized Method Invocation Scheme

The binary tree example of Section 3 employs only a simple form of method invocation, viz synchronous, non-transactional invocation. Hermes/ST provides a generalized method invocation scheme in which a method invocation is the central unit for thread creation, transaction creation and explicit concurrency control specification.

Three types of method invocations are distinguished: synchronous, asynchronous and wait-by-necessity invocations. In a synchronous method invocation, the invoking method is suspended until the invoked method returns a value. In an asynchronous invocation, the invoked method executes independently of the invoking method, within a new thread of control. The invoking method does not wait for a result. In a wait-by-necessity invocation [Car90], the invoked method is executed in a new thread of control and a voucher object is returned to the invoking method, which is not suspended. Sending the message redeem to the voucher object returns the result of the invoked method. This may cause the invoking method to be suspended until the invoked method completes.

Every method invocation, regardless of its type, can create a new transaction. This allows six combinations of method invocations: synchronous invocations that do or do not create a transaction, asynchronous invocations that do or do not create a transaction and wait-by-necessity invocations that do or do not create a transaction. Nested transactions result if a method within a transaction invokes another transaction creating method. For example, a transaction creating synchronous invocation may asynchronously invoke a method that does not create a transaction which in turn could invoke a method using wait-by-necessity that creates a nested transaction.

The Hermes/ST generalized method invocation scheme allows concurrency within transactions without subtransactional overheads. This can be achieved by using non-transaction creating asynchronous or wait-by-necessity invocations within a transaction. Furthermore, ancestor transactions are not suspended while asynchronous descendent transactions execute. This makes transaction creating, wait-by-necessity invocations within transactions possible. The extended invocation scheme is implemented through a novel transaction scheduling mechanism described in [Hum93].

A Hermes/ST method invocation in the generalized scheme is specified by the receiver, the message, its arguments, and optional invocation parameters. If no invocation parameters are specified, defaults are assumed. For example, in the invocation branch deposit:amount to:accountName, the message deposit:to: is sent to the receiver branch with arguments amount and accountName. The invocation is synchronous and does not create a transaction (default) since no explicit invocation parameters are specified. In the invocation branch asynchronously; transactionCreating; deposit:amount to:accountName, the mes-

Smalltalk programmers may wish to note that we utilize the cascading construct (";") for the specification of invocation parameters in an unusual way. This is to provide a concise way of specifying such parameters without having to introduce new syntactic constructs.
sage deposit:to: is also sent to the receiver branch with arguments amount and accountName. This time, however, two invocation parameters are specified: asynchronously and transactionCreating. They specify that the invocation is asynchronous and creates a new transaction.

Method invocations that create a transaction can specify a range of additional parameters. They include mode:, retries: and timeout:.

- Two main transaction modes are distinguished: #abortIfFail (default) and #performIfFail. #abortIfFail specifies that an aborting subtransaction causes its parent transaction to abort. #performIfFail specifies that an aborting transaction does not cause its parent transaction to abort – instead, a specified exception is executed.
- retries allows the specification of how many times to retry a failed transactional method invocation before it is aborted.
- In Hermes/ST, network, node and software failures are not distinguished. Furthermore, Hermes/ST does not prevent or detect deadlocks. Therefore, a timeout mechanism has been chosen to trigger transaction aborts. The specification of timeout values can be critical for the overall performance of a system. Because of the dynamic nature of transaction nesting, it can be hard for a programmer to statically specify a timeout value for a method invocation that creates a transaction. Therefore, Hermes/ST provides accumulative timeouts. Every transaction is assigned a timeout value, either explicitly via the invocation parameter timeout or implicitly via a default value. Whenever a subtransaction starts, the parent transaction’s timeout value is increased by the child’s timeout value. Thus, timeouts accumulate over nested transactions. When a transaction’s timeout value is exceeded, it fails, which may lead to a transaction abort, depending on the specified mode and retries parameters.

Another important Hermes/ST invocation parameter is the lock parameter. lock allows methods to be invoked using type-specific, user-defined concurrency control. Section 5.2 gives a description of such concurrency control specifications.

4.1 Specifying Invocation Parameters in Method Interfaces

Often, particular methods are always invoked with the same invocation parameters. For example, a distributed bank transfer is always invoked transactionally. Hermes/ST allows invocation parameters to be specified as part of the external method interface in the declaration of a method. The syntax is as follows.

---

6 Note that a client object invoking a method on a server object is expected to know the method’s external interface.

7 Smalltalk programmers may wish to note that we have implemented a special invocation parameter compiler that runs over method comments. This way of specifying invocation parameters does not require us to change the Smalltalk method declaration syntax and therefore to modify the Smalltalk compiler.
transfer: amount from: branch1 name: account1 to: branch2 name: account2

```
InvocationScheme
  transactionCreating: true
```

Hermes/ST classes are defined in an inheritance hierarchy. Invocation parameters can be specified for all methods of Hermes/ST classes. When methods are overridden in subclasses, all invocation parameters specified by ancestor classes are inherited individually and can be overridden individually. Invocation parameters for a particular method that are not explicitly specified in the method definition and are not explicitly specified in the definition of the method in any ancestor class are determined by a default value. The external interface of Hermes/ST methods conceptually includes the values for all invocation parameters, determined either by explicit specification, inheritance or default values. Clients that invoke a Hermes/ST method may override invocation parameters specified in its interface.

See the example of a transfer method and auxiliary withdraw and deposit methods in Appendix A.2. The methods deposit:to: and withdraw:from: of class Branch are specified to create a new transaction when invoked. By default, invocations of deposit:to: and withdraw:from: are synchronous. The method transfer:from:name:to:name: is specified to create a new transaction when invoked. It invokes deposit:to: and withdraw:from:, overriding the interface definition from synchronous to asynchronous. When transfer:from:name:to:name: is invoked, a top-level transaction with two asynchronous subtransactions is created.

4.2 Support for Incremental Development

The Hermes/ST generalized method invocation mechanism supports an incremental development scheme for reliable distributed systems particularly well. A system developer can design methods with transactions in mind but implement them non-transactionally first. These non-transactional methods are easier to debug since no underlying transactional system masks software failures. After functional validation of these non-transactional methods, transactions can arbitrarily be put in place to maintain data integrity. This process only requires changing invocation parameters — no structural changes need to be made. The now transactional system can be tested, its performance can be monitored and bottlenecks can be detected. Since transactions are expensive, fine tuning may need to be performed to resolve bottlenecks.

One way of decreasing transactional expense is to cut down transactional nesting depth where possible. Consider the transfer example above. Note that
transfer:from:name:to:name: is always invoked transactionally and the whole transaction should abort if either the withdraw or deposit operation fails. Further note that the transfer transaction is relatively short so that the checkpointing introduced by nested transactions is not necessary. Therefore, for performance reasons, the withdraw and deposit operations should not be performed as subtransactions. An alternative implementation of the transfer method is described in Appendix A.3.

Note that the more efficient implementation of the transfer method is possible because Hermes/ST allows concurrent threads within transactions without using subtransactions. Further note how the method invocation parameters specified in the declarations of withdraw:from: and deposit:to: are overridden by their invocations.

For longer, nested transactions, the probability of success can be increased by using nested transactions. retries: and #performIfFail allow parent transactions to continue when subtransactions fail. Transient failures and deadlocks\(^8\) can be managed through retries. Longer failures can be managed by specifying appropriate compensating actions using #performIfFail.

### 4.3 Reuse Advantages

By separating method invocation parameters from method declarations, the Hermes/ST generalized invocation scheme supports convenient reuse of methods in various contexts. Examples are the withdraw and deposit methods, which create a transaction when invoked directly from a teller machine, and do not create a transaction when invoked from within a transfer.

### 4.4 Comparison to Other Approaches

Argus, Avalon/C++ and Arjuna all use similar syntactic constructs to declare transactions. They provide specific "begin" and "end transaction" commands\(^9\) that are specified in the method code. All three systems allow concurrency within transactions only by generating subtransactions from within some sort of concurrent loop construct\(^10\).

Neither of the three systems provides the level of flexibility and extendibility that Hermes/ST does, with respect to method invocation and transaction semantics. In all three systems, changing a non-transactional method into a transactional one entails changing the method code. A given method cannot conveniently be used both transactionally and non-transactionally.

Coupling thread creation with subtransaction creation has several drawbacks. Firstly, the overheads of subtransactions are considerable, requiring accesses to secondary storage. Secondly, adding transactional semantics to an existing

---

\(^8\) A more effective way of combatting deadlocks is described in Section 5.2.

\(^9\) enter action...end in Argus, start transaction{...} in Avalon/C++ and AtomicAction A; A.Begin();...A.End() in Arjuna.

\(^10\) coenter...end in Argus, costart{...} in Avalon/C++.
multi-threaded application cannot be done in an incremental fashion. Structural changes are necessary where threads are created. This also means that multi-threaded applications cannot be used transactionally and non-transactionally at the same time. In Argus, Avalon/C++ and Arjuna, parent transactions are always suspended while child transactions execute. This not only restricts concurrency but also prevents subtransactional asynchronous and wait-by-necessity like constructs.

The Hermes/ST generalized method invocation scheme provides a higher level of flexibility and extendibility than other transactional distributed programming environments. It is implemented through an advanced synchronization mechanism that includes separate threads and transactions. The mechanism is described in [Hum93].

5 Concurrency Control

5.1 Implicit Concurrency Control

The easiest way for an application developer to prescribe concurrency control in a Hermes/ST application is to use system-defined implicit locking. Hermes/ST methods do not have to be specified as "readers" or "writers". Furthermore there is no need for dedicated lock acquisition code to be included in the specification of a method\textsuperscript{11}.

Implicit locking is currently implemented in Hermes/ST via a mechanism referred to as minimal locking [FHR93b]. Minimal locking achieves high concurrency due to a combination of Hermes/ST's fine-grained object model and read/write locking of individual persistent object instance variables\textsuperscript{12}.

The code for the binary search tree, introduced in Section 3, demonstrates implicit locking. See Appendix A.1. When the class BinTree is instantiated as a persistent object, then all instances of BinTree and BinTreeNode are persistent and concurrency controlled. Implicit locking allows concurrent add"add" and "remove" operations to different parts of the tree. Consider the example tree in Figure 1 containing values 4, 6 and 8. Insertions of the values 2 and 5 as part of different transactions can be performed concurrently since they affect different parts of the tree. The same is true for a concurrent removal of the value 4 and an insertion of the value 7. However, the removal of the value 4 and the insertion of the value 1 cannot be performed concurrently since both operations modify the same part of the tree. Implicit locking delays one of the requested operations until after the other operation's transaction has terminated.

\textsuperscript{11} When and if such code is needed, it can, however, be specified. See Section 5.2.

\textsuperscript{12} Providing such a high concurrency can be expensive. Therefore, minimal locking is currently being refined to a variable locking mechanism that can perform implicit concurrency control on a coarser grain. This coarser-grain locking decreases concurrency but it also decreases scheduling expense and the probability of deadlocks.
Support for Incremental Development. Hermes/ST implicit locking allows the transition from (non-concurrent) volatile objects to fully concurrency controlled persistent objects without changing method definitions or adding concurrency control specifications. However, when explicit concurrency control is desired, e.g. for deadlock avoidance, it can be specified as described in Section 5.2.

Reuse Advantages. The fact that concurrency control is not specified within method declarations allows methods to be conveniently used in a non-concurrent and concurrent context. Again, the binary tree implementation of Section 3 serves as an example.

Comparison to Other Approaches. Argus, Avalon/C++ and Arjuna do not provide implicit locking. In all of these systems, the acquisition of a lock is an explicit part of the operation definition. In Argus an atomic type is accessed via an explicit call to read_lock(atomic.object) or write_lock(atomic.object) [Lis88]. In the Avalon/C++ system, operations that are subclassed from the atomic class acquire read or write locks for the operation through read_lock() and write_lock() methods of the atomic class [DHW88]. Locks so acquired are easily thought of as pertaining to the method. Thus a read only method should acquire a read_lock(), and a method that changes object state should acquire a write_lock(). In Arjuna, classes derived from LockCC\(^\text{13}\) acquire locks through calls to its setlock() method. Thus setlock(new Lock(READ)) acquires a read lock for an operation while setlock(new Lock(WRITE)) acquires a write lock [PS88].

Implicit locking is attractive for the following reasons:

- The first and most obvious is that implicit locking relieves the programmer of the burden of specifying concurrency control. For many data types, implicit locking provides adequate concurrency control for "free".
- Implicitly locked data types are always correctly concurrency controlled. The possibility of concurrency control specification errors is eliminated. Such errors can be hard to identify. Some examples include: declaring an operation

\(^{13}\) LockCC is renamed LockManager in [Shr92].
to be a reader instead of a writer; forgetting to declare a method as a reader or a writer; over-specifying a method because the lock granularity is inappropriate.

Implicit locking, although attractive, is deficient in the following ways.

- Some abstract data types have synchronization constraints that are not expressed by implicit locking. For example, a "get" operation on a bounded buffer\(^{14}\) has to be delayed until after a "put" operation has been performed. Such behaviour is not expressed by implicit locking.
- Implicit locking may introduce deadlock and starvation problems.

These problems are addressed by Hermes/ST explicit locking as described in the following section.

5.2 Explicit Concurrency Control

Hermes/ST explicit concurrency control is achieved through the *programmable lock approach* [FHR93b]. In the Hermes/ST programmable lock approach, type-specific concurrency control is defined in the class specifications of programmable locks. Programmable locks form a hierarchy with the abstract class *ProgrammableLock* as the root. Hermes/ST provides a set of system-defined programmable lock classes. They include classes for mutual exclusion, traditional read/write locking, fair read/write locking and bounded buffer synchronization [FHR93c]. The class *ProgrammableLock* defines two methods, *isSchedulable:* and *isCompatibleWith:*, which return boolean values, in this case true. These methods can be overridden by subclasses. The method *isSchedulable:* allows a programmable lock to make scheduling decisions on the basis of persistent object state. The method *isCompatibleWith:* defines a programmable lock's "compatibility" with other programmable locks.

Programmable locks are associated with Hermes/ST methods via the lock invocation parameter (see Section 4) and instantiated when such a method is invoked. This association allows the specification of parameters that will be passed to a programmable lock. Arbitrary objects can be passed as parameters to a programmable lock. Two types of parameters deserve a special mention. These are the arguments of a method invocation and guard methods.

**Passing Arguments of a Method Invocation to a Programmable Lock.** Consider the example of programmable lock class *AccountWriteLock* which is associated with method *deposit:to:* of class *Branch* (see Appendix A.4). The lock association in *deposit:to:* specifies that the argument *accountName* is passed to *AccountWriteLock*. The argument (accountName) is used by *AccountWriteLock's isCompatibleWith:* method to test whether *otherLock* refers to the same account as the lock itself. This re-defined behaviour of *isCompatibleWith:* only allows *AccountWriteLocks* to be granted over different accounts within one branch.

\(^{14}\) A bounded buffer is a fixed size, first-in first-out (FIFO) queue.
Passing Guard Methods to a Programmable Lock. Guard Methods [Atk91] are read-only methods that allow programmable locks to inspect object state. Consider the example of programmable lock class SavingsAccountsWriteLock which is associated with method addInterest of class Branch (see Appendix A.5). addInterest accesses all savings accounts of a branch to add any outstanding interest. SavingsAccountsWriteLock conceptually locks all savings accounts of a branch in write mode to allow addInterest to be performed atomically. SavingsAccountsWriteLock isCompatibleWith: checks the type of otherLock's account (#cheque or #savings) using the guard method typeCheckMethod. This guard method is passed as a parameter to SavingsAccountsWriteLock in the lock invocation parameter specification of method addInterest.

Using Programmable Locks for Deadlock Avoidance. Hermes/ST implicit locking may cause deadlock if, for example, a branch-internal transfer operation from one savings account to another savings account interferes with and addInterest invocation. Associating addInterest with a SavingsAccountsWriteLock and associating withdraw:from: and deposit:to: (the two methods invoked in a transfer operation) with an AccountWriteLock avoids such a deadlock. This is because SavingsAccountsWriteLock conceptually locks all savings accounts of a particular branch in write mode. A SavingsAccountsWriteLock is incompatible with every AccountWriteLock that controls the access to a savings account. Thus, in case of a conflict, the execution of one of the operations (transfer or addInterest) is delayed until after the other operation's transaction has terminated (committed or aborted).

Support for Incremental Development. If it is necessary to add explicit concurrency control to an implicitly concurrency controlled Hermes/ST application, the incremental strategy still applies. First, simple system-defined programmable locks like mutual exclusion locks or read/write locks can be employed. Performance analysis of the simply concurrency controlled system may detect bottlenecks. These bottlenecks can then be alleviated by the introduction of more sophisticated application-specific programmable locks such as SavingsAccountsWriteLock and AccountWriteLock.

Reuse Advantages. The Hermes/ST explicit concurrency control mechanism does not only support the reuse of methods that are explicitly concurrency controlled. It also facilitates reuse of concurrency control specifications themselves.

- The association of programmable locks and Hermes/ST methods is separated from the method definition. This allows one to conveniently use a method in a sequential and concurrent context.
- The concurrency control specification for a Hermes/ST class is composable: subclasses that add and/or override methods can individually add/change programmable lock associations. Composability is achieved by a combination
of separating the programmable lock association from method definition and associating programmable locks with methods individually.

- Programmable locks are specified separately from the Hermes/ST classes in which they are applied. This allows a common concurrency control behaviour (e.g. mutual exclusion) to be applied in different classes where appropriate.
- Since programmable locks are defined in an inheritance hierarchy, concurrency control behaviour can be reused through "programming by difference". Examples are the implementations of SavingsAccountsWriteLock and AccountWriteLock which utilize the locking behaviour of their superclass WriteLock and weaken the compatibility predicate using a logical "or" operator.

Comparison to Other Approaches. Argus, Avalon/C++ and Arjuna all support user-defined concurrency control. Type-specific concurrency control in Argus [WL85] has a different goal to its counterpart in Hermes/ST. The goal of Argus' user-defined atomic types is to permit higher concurrency than strict two phase locking allows. One goal of Hermes/ST's user-defined programmable locking is to further restrict concurrency allowed by implicit locking in order to avoid problems such as deadlock and starvation. Therefore, the mechanisms are not further compared.

Similar arguments apply to Avalon/C++. However, some aspects of Avalon/C++'s approach to user-defined locking [DHW88] do compare with the Hermes/ST approach. The idea that locks are specified via inheritance is shared. Avalon/C++ provides the subatomic class as a starting point for defining a user-defined hierarchy of locks. This use of inheritance is analogous with Hermes/ST programmable lock inheritance. However, since method declarations contain concurrency control information in Avalon/C++, it lacks composability and does not allow classes to be reused sequentially.

User-defined locking in Arjuna [PS88] is similar to the Hermes/ST programmable lock approach. The lock concurrency controller class LockCC exports operations setlock and releaslock. releaslock is called implicitly at transaction termination time. An application calls setlock which then calls lockconflict which in turn calls the != operator. The != operator is analogous to the isCompatible: method in the Hermes/ST programmable lock approach. It can be overridden in user-defined locks.

Arjuna, like Hermes/ST, permits object state to be passed to locks during the instance creation of a lock. However, it does not support inspection of object state through guard methods. Thus, it is not clear how an operation such as a bounded buffer "get" can be specified.

Consistent with programmable locks in Hermes/ST, locks in Arjuna are organised in an inheritance hierarchy and are specified independently of their use. Thus Arjuna's user-defined lock specifications can be re-used and can be extended via inheritance. Locks, however, are not associated with a method but are a part of the method definition. Thus, concurrency specifications are not composable. Therefore, Arjuna lacks some of the reuse advantages that Hermes/ST provides.
6 The Banking Example in Hermes/ST

This section discusses some method definitions of the main classes Branch (Appendix A.6) and Teller (Appendix A.7) as coded in the banking example, introduced in Section 2. The class definition for Branch specifies an instance variable accounts which is initialized to an empty persistent binary tree in the Branch instance creation method (not shown). All accounts contained in a particular branch are stored in accounts, ordered according to their accountName.

Methods like deposit:to: and withdraw:from: (not shown) use an auxiliary method lookUp:. lookUp: descends the accounts tree to return a Hermes/ST object reference to the specified account. In the case that the account cannot be found, abortCurrentTransaction: is invoked. In the case of a transactional invocation, this causes the current transaction to abort and the specified symbol #noSuchAccount to be passed to the client of the aborting transaction. In the case of a non-transactional invocation, an exception is raised. Methods deposit:to: and addInterest are explicitly concurrency controlled using programmable lock classes AccountWriteLock and SavingsAccountsWriteLock, as described in Section 5.2.

The class definition for Teller specifies an instance variable interface, which is initialized to a volatile graphical user interface whenever a teller is started up (e.g. after a node crash). This volatility is specified in an initialization method (not shown).

The method transfer:from:name:to:name: performs a traditional fund transfer with the optimization described in Section 4.2. The method internationalTransferFrom:name:to:name: implements a more complex international transfer operation that involves a currency exchange. This method is interesting since it uses all three types of method invocations, viz synchronous, asynchronous and wait-by-necessity. Assume that every branch keeps a currency table for all traded currencies which might be slightly out of date. A currency table which always keeps the exact current exchange rate can be remotely accessed at the head office. Assume that for small transfers, i.e. transfers that do not exceed a particular limit, the locally stored exchange rate can be used whereas for large transfers, the exact rate must be used. In order to optimize the performance of the transfer method, the exchange rate request to the head office is performed concurrently with the amount request to the source branch — using a wait-by-necessity and a synchronous invocation. If the amount to transfer does not exceed the limit, then the actual transfer can go ahead without waiting for the exact exchange rate to be returned. The voucher exactRate is only redeemed when necessary. The actual transfer is performed concurrently using asynchronous invocations with the optimization described in Section 4.2.
7 Conclusions

In this paper, linguistic features of the Hermes/ST distributed programming environment have been presented. It has been shown that parameterisation in Hermes/ST leads to extendibility and flexibility of reliable distributed applications. These support incremental development and reuse. This has been demonstrated throughout the paper with reference to a simple distributed application, a distributed banking system. This paper has concentrated on Hermes/ST's linguistic features rather than on their implementation details. The implementation of the mechanisms to support the Hermes/ST distributed programming model are described in [FHR93a, Hum93, FHR93c].

All of the code presented in this paper has been compiled, tested and run on a Hermes/ST installation over a cluster of work stations linked via a local area network. Furthermore the development techniques proposed were successfully utilised during the development of the banking application.

Future extensions to the Hermes/ST distributed programming model will be directed at attempting to parameterize more features of the distributed programming environment. The current implementation supports volatile objects and persistent objects only. The introduction of other kinds of objects is currently being performed. Similarly, an investigation into the usefulness of parameterizing serializability, atomicity and persistence in the process model is being undertaken. Thus the application developer will have a broader process choice than just "transactional" or "non-transactional".

References


GR89] Adele Goldberg and Dan Robson. Smalltalk-80: The Language. Addison-Wesley, 1989.


A Hermes/ST Code Examples

A.1 Instance Methods for BinTree

class name BinTree
superclass HermesCollection
instance variable names root

Class methods

instantiate: kind withContents: anObject
"create a new binary tree instance of the specified kind,
initializing the root node to refer to anObject”

\[ \text{inst} := \text{super instantiate: kind.} \]
\[ \text{inst root: (BinTreeNode instantiate: kind withContents: anObject).} \]
\[ \text{\textasciitilde inst} \]

\textit{Instance methods}

\textbf{add: anObject ifExisting: aBlock}

\textit{Add anObject to the binary tree. If the object already exists, execute aBlock instead.}

\[
\text{if self root isNil}
\text{ifTrue: [self root: (BinTreeNode instantiate: self kind withContents: anObject)]}
\text{ifFalse: [self root add: anObject ifExisting: aBlock].}
\]

\textbf{remove: anObject ifAbsent: aBlock}

\textit{Remove anObject from the binary tree, and execute aBlock if it didn’t exist in the tree.”}

\[
\text{if self root isEmpty ifFalse: [self root contents = anObject}
\text{ifTrue: [self root right isNil}
\text{ifTrue: [(nextNode := self root left) isNil}
\text{ifTrue: [self delete: self root.
\text{self root: nil]}
\text{ifFalse: [nodeToRemove := self root.
\text{self root: nextNode.
\text{self delete: nodeToRemove]]}
\text{ifFalse: [self root contents: self root removeLeastFromRightSubtree]]}
\text{ifFalse: [self root remove: anObject ifAbsent: aBlock]]}
\text{ifTrue: [aBlock value]}
\]

\textbf{A.2 Invocation Parameters for Transfer Methods}

\textit{Teller instance methods for teller operations}

\textbf{transfer: amount from: branch1 name: account1 to: branch2 name: account2}

\textit{\textasciitilde}
InvocationScheme
  transactionCreating: true
"

branch1 asynchronously; withdraw: amount from: account1.
branch2 asynchronously; deposit: amount to: account2.
^#done

Branch instance methods for account operations

deposit: amount to: accountName
  
  InvocationScheme
  lock: [AccountWriteLock account: accountName]
  transactionCreating: true
"

...

withdraw: amount from: accountName
  
  InvocationScheme
  lock: [AccountWriteLock account: accountName]
  transactionCreating: true
"

...

A.3 A More Efficient Implementation of the Transfer Method

transfer: amount from: branch1 name: account1
to: branch2 name: account2
  
  InvocationScheme
  transactionCreating: true
"

branch1 asynchronously; nonTransactionCreating;
  withdraw: amount from: account1.
branch2 asynchronously; nonTransactionCreating;
  deposit: amount to: account2.
^#done
A.4 Definition and Usage of AccountWriteLock

class name: AccountWriteLock
superclass: WriteLock
instance variable names: account

isCompatibleWith: otherLock
  "(super isCompatibleWith: otherLock)
  or: [self account ~= otherLock account"

Branch Protocol for account operations

deposit: amount to: accountName
  "
  InvocationScheme
  lock: [AccountWriteLock account: accountName]
  transactionCreating: true"

A.5 Definition and Usage of SavingsAccountsWriteLock

class name: SavingsAccountsWriteLock
superclass: WriteLock
instance variable names: account, typeCheckMethod

isCompatibleWith: otherLock
  "(super isCompatibleWith: otherLock)
  or: [(self
    performGuard: self typeCheckMethod
    with: otherLock account)
    = #cheque]

Branch Protocol for account operations

addInterest
  "
  InvocationScheme
  lock: [SavingsAccountsWriteLock
    account: #allSavingAccounts
    typeCheckMethod: #typeOf:]
  transactionCreating: true"
A.6 Hermes/ST Class Branch

class name Branch
superclass Root
instance variable names name accounts

Protocol for account operations

lookUp: accountName

```
self accounts detect: [:account | account name = accountName]
ifNone: [self abortCurrentTransaction: #noSuchAccount]
```

deposit: amount to: accountName

```
InvocationScheme
lock: [AccountWriteLock account: accountName]
transactionCreating: true
```

| account |
amount < 0 ifTrue: [self abortCurrentTransaction: #negativeAmount].
account := self lookUp: accountName.
account balance: account balance + amount.
^#done

addInterest

```
InvocationScheme
lock: [SavingsAccountsWriteLock
account: #allSavingAccounts
typeCheckMethod: #typeOf:]
transactionCreating: true
```

self accounts do: [:account | account type = #savings
ifTrue: [account balance: account balance * 1.025]].
^#done

A.7 Hermes/ST Class Teller

class name Teller
superclass Root
instance variable names name currencyTable interface

Protocol for teller operations

transfer: amount from: branch1 name: account1 to: branch2 name:
account2

InvocationScheme
  transactionCreating: true

branch1 asynchronously; nonTransactionCreating;
  withdraw: amount from: account1.
branch2 asynchronously; nonTransactionCreating;
  deposit: amount to: account2.
^#done

internationalTransferFrom: branch1 name: account1
to: branch2 name: account2

InvocationScheme
  transactionCreating: true

| currency1 currency2 exactRate amount newAmount |
currency1 := self currencyOf: branch1.
exactRate := self headOffice waitByNec;
  exchangeRate: currency1 to: currency2.
amount := branch1 balanceOf: account1.
newAmount := amount * (amount > 10000
  ifTrue: [exactRate redeem]
  ifFalse: [self exchangeRate: currency1 to: currency2]).
branch1 asynchronously; nonTransactionCreating;
  withdraw: amount from: account1.
branch2 asynchronously; nonTransactionCreating;
  deposit: newAmount to: account2.
^#done