A Framework for Transactional Semantics

R. David Ranson
Telecommunications Software Research Centre (TSRC)
University of Wollongong, Department of Computer Science
P.O. Box 1144, Wollongong NSW 2522, Australia.
Phone: +61 42 268832; Fax +61 42 273277.
Email: dranson@cs.uow.edu.au

Abstract

This paper introduces a framework in which nested transactional and less-than-transactional semantics can be defined and specialized. The less-than-transactional semantics are based on the transactional properties of isolation, atomicity and permanence. These properties are complemented by meta properties expressing their scope, their obligations if nested, and their relationship if combined. The framework is employed to define a useful set of such semantics. These are modified within the framework to provide a further set of semantics. This approach integrates the strong consistency but significant expense and restrictiveness of transactional semantics with the needs of a broader range of applications which require less-than-transactional semantics. Applications can choose an appropriate level of efficiency and robustness. Application porgrammers can refine the semantics provided to better suit their system. The breakup of these properties also encourages the reuse, maintainability and development of such applications. These semantics are supported in a distributed object-oriented programming environment, Hermes/ST.

1 Introduction

Transactions, as adapted from the database community, have proven to be a useful abstraction for achieving consistency and robust computation in distributed systems which are subject to partial failures. Transactions are a process abstraction, providing *isolation*, *durability* and *atomicity* properties for actions over potentially distributed objects [Dav73, Gra78]. The isolation property specifies that a transaction will appear to have executed without interference despite the possible existence of concurrent transactions. The atomicity property specifies that the transaction will execute in its entirety, or appear to have not executed at all. The durability property specifies that the effects of a transaction will outlast subsequent node failures.

In a distributed object-oriented system, objects usually require certain properties in order to support accesses by transactions. These properties are *concurrency control*, to isolate concurrent accesses, *permanence*, to provide durability in the

face of node failure, and *undoability*, to allow recovery from failed accesses. However, in general purpose computing environments, there exist objects which cannot or should not support these properties. For these, objects, transactional properties are too restrictive. There also exist applications where the robustness that transactions provide is not required, and where the added expense of transactional overheads is undesirable.

Most existing distributed transactional systems have attempted to address these deficiencies through the provision of various less-than-transactional semantics. These include the relaxation or complete removal of transactional properties. Argus [Lis88], for example, permits transactions to spawn top-level transactions from within transactions, and to access non-atomic data. These relaxations can lead to violations of a transaction's isolation, atomicity and durability properties. They do, however, allow a wide range of cheaper and less-than-transactional semantics to be applied, but in a rather uncontrolled fashion. In many systems, such as in Argus, Avalon/C++ [EME91] and Venari/ML [HKM+94], convention is often employed to maintain the semantics of transactions in the presence of these less-than-transactional facilities. In open systems, however, this approach may not be sufficient to satisfy the requirements of clients requiring strict transactional semantics from their encapsulated servers.

The main aim of this research is to address these deficiencies more directly. A distributed programming environment should provide a broader specification language over method invocations and objects, to allow distributed nested transactions as well as less-than-transactional properties. At the same time, the system should provide stronger support when required for the properties of strict transactions, where they interact with other less-than-transactional abstractions. This may mean strengthening the properties of the less-than-transactional abstractions. Ideally, the system should be capable of specifying most of the more useful less-than-transactional semantics provided by current transactional systems [Lis88, EME91, HKM+94, FAC+94, CD89, HMSC88]. To achieve this diversity, the system should allow some facility for the specialization of any default semantics provided.

Another aim of the research is to maintain support for composability, incremental development, reusability, and maintainability, as was outlined in [FHR94]. These mechanisms prevent class explosions, allowing classes to be instantiated as transaction supporting or volatile without any change in class code.

Examples of the semantics which the system should support include distributed nested transactions, isolated threads, synchronised threads, durable threads, and non-durable but isolated and atomic threads.

These aims are being addressed in an extension to Hermes/ST, a distributed object-oriented programming environment [FHR94, Hum93, FHR93a, FHR93b]. This extension supports the specification of transactional and less-than-transactional semantics over messages, as well as the specification of transaction-supporting and less-than-transaction supporting properties over objects. Each is more suitable in certain circumstances. In addition, the extension presents mechanisms to resolve differences between message property requirements and object property support. The less-than-transactional semantics provided are based on the essentially independent transactional properties of isolation, durability and atomicity. Hermes/ST

thus represents an attempt at a smoother and more controlled integration of purely transactional systems with broader systems, often requiring less-than-transactional process abstractions over objects with less-than-transaction-supporting properties. The varying message semantics provided are supported in a uniform manner.

Using these semantics, an application programmer can achieve better performance in many applications, by taking advantage of situations where full transactional semantics are not required, are unnecessarily restrictive, or are simply not possible. At the same time, the application programmer is able to specify and achieve full nested transactional semantics where these are required, and remain confident of their adherence to by subsystems.

In this paper, a framework is described by which the process semantics for the system are defined. The framework is based on the identification of a set of base and meta-properties, the sum of which is sufficient to describe the semantics of nested transactions. The base properties are Atomic, Isolated and Durable. The meta-properties are Nesting, Scope, and Order. This framework enables the definition of semantics for both nested transactional and less-than-transactional abstractions. Furthermore, the framework supports the redefinition and customization of these semantics, possibly by application or system programmers. Current transactional systems do not provide such support for the definition and customization of less-than-transactional semantics.

The paper is structured as follows. Section 2 introduces the framework used to define the required semantics. This also describes system support for the framework. Section 3 describes the way in which the framework was filled to define a semantics for the current implementation. This section also describes ways in which these default semantics can be modified or specialized. Section 4 outlines some efficiency considerations regarding the implementation. Section 5 compares the system with existing approaches. Section 6 concludes the paper and describes current and future work.

2 The Semantics Framework

To achieve the aim of expressing a diverse range of semantics, a framework has been developed which allows the definition and specialization of semantics for nested transactional and less-than-transactional abstractions. These semantics are process-based. The framework thus defines a set of semantics which are expressed most naturally over *messages* (method invocations). Tree terminology is used to describe the relationship between a message (parent/ancestor) and the messages it invokes (children/descendants/submessages).

As a foundation for the framework, several essentially independent properties have been identified. These properties have been deemed to be sufficient to define the semantics of messages in a nested transactional system. These are the base properties of transactions, viz. Atomic, Durable and Isolated, together with meta-properties of Nesting, Scope, and Order. The base properties provide useful semantics both individually and in combination. A base message property's semantics are defined through its pre, post, and fail methods, which access system-provided base mechanisms. These definitions may be extended for each base property through

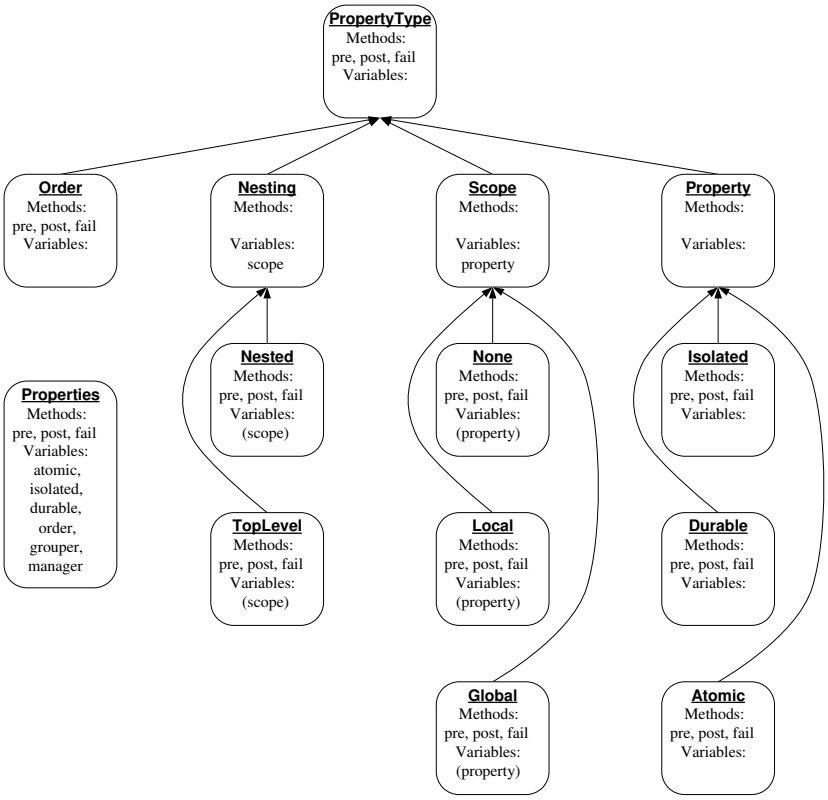

Figure 1: The Semantics Framework

similar methods in meta-properties Nesting, Scope and Order. Scope defines the extent of a message property in terms of itself and its descendant messages (its requirements of messages called as a part of it). Nesting defines the semantics of a property when nested under the same property (its possible obligations as a part of an ancestor message). Finally, Order allows the specification of the order in which base property methods are executed.

The framework specifies what must be defined for the properties and the position of these properties in relation to each other. The framework is structured using object-oriented principles (inheritance in particular), as is shown in Figure 1. Arrows indicate an inheritance class relationship.

The semantics of the various properties are defined in classes, all of which are subclassed from the abstract superclass, PropertyType. PropertyType specifies three empty instance methods: pre; post; and fail. These are overridden in subclasses. pre specifies the processing required to implement a property prior to message execution (e.g. acquire lock in the case of the property Isolated). Similarly, post specifies the processing required following message *completion* i.e. following the execution of the last line of code, or a return statement. post processing may defer activity until the completion of an ancestor message. fail specifies the processing required in the event of message failure.

In the current implementation, a Message instance describes a method invocation. This refers to an instance of Properties, a properties grouping class. This has references to each base property via their meta-properties Nesting and Scope, as well as references to Order. It also has references to implementation-specific concerns,

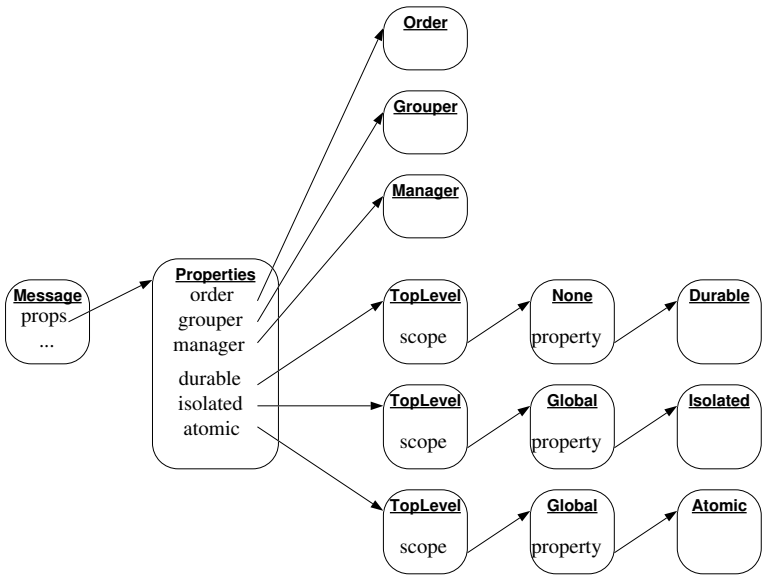

Figure 2: A Global-Isolated Global-Atomic None-Durable Message

such as a MessageManager and a meta-message Grouper (see Figure 2).

2.1 Base Message Properties

In a general sense, a durable message's effects are intended to outlast node failures. Its changes are made permanent by being mirrored on secondary store. An atomic message should ensure that either all or none of the message's effects are performed. A failed atomic message should have no effects on the system. An isolated message should not interleave access to a resource or resources with other messages. It acquires locks over the resources it accesses, and releases these at message completion or later.

The precise definition of these base properties is specified in the corresponding framework Property subclasses, which call system-provided base property mechanisms. These base mechanisms are intrinsic to the framework, and their definition is not readily accessible or changeable by an application programmer.

Permanence for durable messages is provided through logging to secondary store. The log manager interface provides asynchronous and synchronous writing of data to the log. Asynchronous writes (writeAsynchronously) cause the log manager to write data to its cache, and to actually write the cache to secondary store as and when it sees fit. This allows greater efficiencies in writing changes to secondary store. A synchronous write (writeAndFlush) ensures that the data is saved to secondary store prior to the synchronous write method's return. It also flushes (flush) any changes buffered in the log manager's cache. The log is flushed from time to time, whereby only the most recent complete versions of objects are retained. A complete version of an object is achieved at transaction commit, in the case of transactional changes, or following a successful log cache flush in the case of non-transactional durable changes. Following a node crash, permanent objects are recovered from the log and restored when appropriate to memory.

Concurrency control is as specified for Hermes/ST [FHR93a], which provides

system defined concurrency controllers as well as a facility for the writing of type-specific concurrency controllers. This concurrency control is pessimistic, relying on the acquisition (or inheritance) of locks prior to method execution (setLock), and their release (or anti-inheritance) at some point following method completion (releaseLock). *Lazy-evaluation anti-inheritance* is employed [Lis84]. Deadlock and failure detection are provided through *timeouts*, which fail a message through an exception mechanism. System defined concurrency control provides read/write locking for methods over objects. Moss' locking rules, as specified for nested transactions, are employed for all nested global-isolated messages [Mos85].

Atomicity is provided by a state restoration handler. This allows the copying of old data prior to its being changed (pushStack). Nested atomic invocations create a *version stack* of old data [Ls83]. Successful message completion may cause the old values to be discarded (popStack). If an atomic message fails via an exception, then old values may be restored (restoreAndPopStack) from this in-memory copy (according to the fail method definitions).

The degree to which these base mechanisms have been exposed in the framework has been based upon the perceived benefits of their exposure (in terms of the additional useful semantics expression). A secondary concern has been the difficulty in exposing and allowing customization of these mechanisms. A typical example of a partly exposed mechanism, supported elsewhere in the Hermes system, is concurrency control. Hermes/ST provides implicit, fixed, system-defined concurrency control through read/write locking. However, the application programmer is also provided with a mechanism to customise the concurrency control for specific classes. Conversely, the complexity of the message scheduler was deemed to be too great to present much benefit from its exposure.

Messages can have combinations of these base properties. The framework ensures that each of these combinations has a well-defined semantics. For example, the Atomic, Durable and Isolated properties together provide transactional semantics. Some of these combinations, however, have useful meaning only in restricted contexts (e.g. non-isolated messages are useful in non-concurrent or concurrent read-only contexts, but are not generally useful in concurrent reading/writing contexts).

The base message properties, have been described as being orthogonal [FAC+94]. This allows for their essentially modular implementation, as well as for their arbitrary composability. Providing semantics for these properties both individually and in combination supports incremental development (through the ability to gradually strengthen or weaken message properties), as well as class reuse and maintainability.

2.2 Scope of Message Properties

Scope is a meta-property which defines the extent of a message property. The framework distinguishes three types of Scope. Each message property is defined as being either Global, Local or None in scope.

A None message property means that the message does not have that property.

A Local message property effects the receiving object only. This property has no influence over any descendant messages.

A Global message property effects all objects visited by the message and by any descendant messages of the message. This property constrains all descendant messages to have at least the same property with Local scope.

This allows the definition of semantics which might, for example, cause the sychronisation of concurrent accesses to individual objects, while interleaving those accesses within groups of objects (local-isolated). This is analogous to short-duration semaphores over individual objects. Alternatively, two globally isolated messages are *serialized* [BHG87]:- message accesses to the same group of objects will appear as if they occurred in some serial order. This particular semantics opens messages up to the possibility of deadlock, the management of which is described in Section 2.1.

2.3 Nesting of Message Properties

Nesting is a meta-property which defines the obligations that related nested and top level messages have toward each other. This influences the semantics of a message property. The framework distinguishes two types of Nesting. Each message property is defined as being either TopLevel or Nested.

A message with a TopLevel meta-property has no recorded ancestor message with the same global property.

A Nested message has some TopLevel ancestor message with the same global property.

Nesting is usually determined by the system, based on a message's ancestor properties. A message's parent refers to the closest ancestor with the same global property.

In Moss' model of nested transactions [Mos85], for example, a nested transaction defers its post-message-execution activity until the commit of its top-level transaction. A descendant (nested) message of a globally atomic message should maintain undoability until the top level global-atomic message has completed, in order to preserve the semantics of the global atomicity of the top level message. Its changes should be visible upon completion to other descendant messages of the top level atomic message, but they should not be visible to unrelated messages. Similar semantics usually apply to messages nested under global-isolated messages (which don't release locks until top-level global-isolated message completion), and to messages nested under global-durable messages.

2.4 Order

Implicit to the framework, an order is defined over the execution of meta-properties: Nesting is evaluated before Scope. The order in which base properties should be evaluated, however, is not fixed. In fact, the particular order of base property evaluation is important to semantics definition, and thus requires explicit definition.

Order is a meta-property which defines an order over the execution of the pre, post and fail methods of method invocations employing more than one base message property.

For example, it is usual for locks to be acquired prior to recoverable copies being made in an atomic isolated message (Isolated pre before Atomic pre). The same message might also require that locks be released after a copy is restored (Isolated fail after Atomic fail), yet locks may be released if desired prior to old copies being discarded (Isolated post before Atomic post).

2.5 Property Strictness

The base mechanisms described in Section 2.1 are expressed over objects as instance properties: Concurrency Control, Permanence, and Undoability. This is often easier, more appropriate and safer than through some convention or rules over the object's method invocation properties. For these reasons, the system supports the specification of transaction and less-than-transaction supporting properties over objects. For example, views (representing screen windows) should generally not be permanent, but as a result do not guarantee transactional properties.

This facility leads to the possibility of a conflict between the properties a message may require, and those that an object can provide. Similar conflicts arise between messages with incompatible properties. For example, how should a synchronised or globally isolated message interact with a non-isolated message? Finally, messages with certain properties may conflict with the property requirements of ancestor global messages.

These conflicts are resolved through the Strictness construct. Each base object and message property may be further characterised by a boolean indicating its Strictness. A non-strict property will attempt to adapt to conflicting properties by assuming the stronger and less-efficient semantics, while a strict property will not. Table 1 describes actions to be taken on conflicts between message and message or object properties.

	Strict	NonStrict
Strict	Error	Elevate Property
NonStrict	Elevate Property	Elevate Property

Table 1: Resolving Conflicting Properties

3 Filling the Framework

Tables 2, 3, 4 and 5 demonstrate how a set of useful semantics can be defined using the framework. Methods are presented using pseudo code only. This definition of the methods, and the ensuing semantics represent a validation of the framework. The semantics defined in the tables are driving a current design and implementation, which is based on the class hierarchy shown. This is being implemented in Smalltalk-80 [GR89], as an extension to Hermes/ST.

In an implementation, a message manager is required to store and act on information relating to submessages (children and descendants). This collects information such as descendant message creation, completion, and objects visited,

which is required by meta-property post and fail methods. The system defines MessageManager behaviour, including methods addChild, nowFinished, subGroupFailed, and childFailed. A global parent message can manage a global submessage failure (subGroupFailed) in many ways. A global parent message must fail, however, if a non-global child message fails (childFailed). Messages contain a messagePath, which is used to describe a message's ancestor messages, as well as their properties.

3.1 What are the Semantics Provided?

Tables 2, 3, 4 and 5 define a semantics for nested transactional and less-than-transactional properties. For example, a top level global-atomic global-isolated global-durable method invocation provides (non-distributed) transactional semantics. Submessages acquire nested local-atomic, local-isolated and local-durable properties (see Section 2.3) unless otherwise specified. They inform their parent when they are about to begin (Nested pre), then acquire locks (Isolated pre) and maintain undoability (Atomic pre). Submessage completion involves no more than informing their parent of the message completion. A global submessage property (such as in a nested transaction), must first wait until all children are finished, before informing its parent. When the top-level message has completed execution, it waits until all activity of submessages has completed (Global post), and then begins to synchronously save changes to the log (Durable post), clean up state restoration handlers (Atomic post), and finally to release locks (Isolated post). This implements two-phase locking [EGLT76]. The fail methods define transaction abort, which undo all changes made by the transaction, and release its locks.

	Nested	TopLevel
pre	parent addChild; Scope pre	Scope pre
post	if global then (wait until all children finished); parent NowFinished	Scope post
fail	if global then (Scope fail; parent subGroupFailed) else (parent childFailed)	Scope fail

Table 2: Nesting subclass method definitions

	None	Local	Global
pre	return	Property pre	Property pre
post	return	Property post	wait until all children finished; children Scope post; Property post
fail	raise exception	Property fail	children Scope fail; Property fail

Table 3: Scope subclass method definitions

A global-atomic global-isolated global-durable method invocation called as a part of the above transactional message acquires the traditional semantics of a nested transaction. This allows failure of the subtransaction (Nested fail $\rightarrow$ Global fail ... $\rightarrow$ parent subGroupFailed) without necessarily aborting the top-level transaction.

	AtomicRead	AtomicWrite	Durable	Isolated
pre	return	pushStack	return	setLock
post	return	popStack	writeAndFlush state	releaseLock
fail	return	restoreAndPopStack	writeAndFlush 'failed'	releaseLock

Table 4: Property subclass method definitions

	Order
pre	isolated pre; atomic pre; durable pre
post	atomic post; durable post; isolated post
fail	atomic post; durable post; isolated post

Table 5: Order method definitions

A top level global-isolated non-atomic non-durable method invocation provides the semantics of a serialized message. This executes in isolation from all other non-descendant messages, yet its changes are not saved to disk, nor are they undoable. This allows a message to access several objects without interference or interleaving with other messages. Unless governed by some deadlock prevention mechanism, this message is, however, subject to deadlock, whereupon the message will fail, leaving the objects in whatever state they were in at deadlock. A message which wishes to maintain certain consistency predicates between several objects could instead be global-isolated global-atomic. This is shown in Figure 2, in which arrows indicate instance references. This would undo any changes made in the event of failure, as possibly caused by deadlock, thereby maintaining inter-object consistency. Such messages are considerably less expensive than equivalent transactional messages, which also require that all of the objects visited be persistent.

A top level local-isolated method invocation in the above definition provides intra-object consistency (sychronization). Its changes to its receiver object are not interfered with by other concurrent messages, while submessage changes to the same or other objects are concurrency controlled according to their own properties. This semantics is again considerably less expensive to implement, and allows greater concurrency than global-isolated or transactional messages.

3.2 Modifying the Default Semantics

The semantics defined in Tables 2, 3, 4 and 5 may be modified through specialization to provide a far greater range of semantics. In this way, the semantics may go closer to providing what has been proposed throughout the literature for nested transactions and less-than-transactional properties. Examples of these different semantics which one might wish to support include *lazy saving*, *nested transactional checkpointing*, and *two phase commit for distributed transactions*. Tables 6, 7 and 8 show the modifications required to provide these additional semantics.

Lazy saving, which is akin to what is provided by traditional UNIX^{TM} file systems, is provided by specializing Durable to LazyDurable. This subclass overrides the post method, to write asynchronously to the log. These changes may not yet exist on the disk at the time of the return of the message.

	LazyDurable
post	writeAsynchronously state

Table 6: LazyDurable

Nested transactional checkpointing, or *early writing*, was suggested by Moss as a means of managing partial failures within transactions. In this semantics, the changes made by a nested transaction are saved to disk at subtransaction commit rather than only at top level commit. The top level transaction can rely on the permanence of these changes (and possibly proceed to commit even if the subtransactional node has failed), or can still abort the changes saved by the subtransaction. To effect this change, a subclass of Nested, NestedEarlyWrite, can override the post method of Nested to also call the post method of its Scope and Property. A message employing the definitions of NestedEarlyWrite as its Nesting meta-property for its Durable property, would acquire the semantics of nested transactional checkpointing.

	NestedEarlyWrite
post	if global then (wait until children finished; Scope post; parent NowFinished)

Table 7: NestedEarlyWrite

Distributed transactions require a two-phase commit rather than the single phase commit that is sufficient for a single node transaction. This can be provided by a specialization of the class Properties. The subclass TopLevelDistributedTransaction overrides the post method of Properties to implement the two-phase commit protocol.

	TopLevelDistributedTransaction
post	when all children completed (children inParallelDoAndWait: (durable post)); writeAndFlush 'commit'; children do: (writeAndFlush 'commit'; atomic post; isolated post)

Table 8: TopLevelDistributedTransaction

4 Efficiency Considerations

An implementation based directly on the conceptual classes of the framework provides easy to implement and correct semantics, but at the cost of efficiency. For example, in the case of distributed transactions, several (meta) messages must be passed between possibly remote submessage meta-properties and their parent message manager i.e. the Isolated Nesting's pre informs the parent transaction, followed by the Atomic Nesting's pre informing the same parent transaction, etc).

Current implementations of distributed nested transactions do not suffer from such expensive repetition. An implementation of meta-property message grouping, which overcomes this inefficiency, is provided. In addition to this, rather than instantiating up to 12 objects to describe message properties for each message, the semantics of the properties can be recomposed or compiled into more efficient implementations.

5 Comparison with Other Work

The author is not aware of any other work that provides a framework for the definition and customisation of nested transactional and less-than-transactional properties. As stated in the introduction, several systems exist which provide distributed nested transactional semantics. Many of these provide ad hoc semantics for avoiding the expense and sometimes unsuitability of transactional semantics [Lis88, EME91].

Some systems provide a breakdown of transactional properties similar to the proposed implementation. The semantics of the properties provided are not customizable, however. Venari/ML provides support for atomic and durable functions, with these base properties being composable [HKM+94]. Such functions are always serialized (globally isolated) if they access non-private data. Venari/ML does not allow different semantics, such as nested-transactional checkpointing or lazy writing. It does not support distribution.

Raven [FAC+94] provides object properties similar to the proposed implementation of Hermes/ST. It supports atomic, durable, lazy-durable and concurrency-controlled object properties whose implementations cannot be modified. Users are allowed to define their own properties, though these cannot access system state. Hence their effects are identical to calling actual object methods.

Clouds [CD89] provides transactions and single-object consistency preserving actions (equivalent to non-isolated, non-atomic, local-lazy-durable) over persistent, concurrency controlled and recoverable objects.

Other work has attempted to meet the performance problems of transactions by loosening the semantics of some of their properties [BK91], though generally without removing properties altogether, as is possible in Hermes/ST. These approaches are directed at an attempt to improve the performance of purely transactional systems, rather than to propose a significantly different range of semantics for these systems.

6 Conclusions and Future Work

This paper has introduced a framework for the definition and specialization of distributed nested transactional and less-than-transactional semantics. This has been used to define a set of new semantics over messages in the Hermes/ST distributed programming environment. This work represents an extension to the Hermes/ST system described in [FHR94, Hum93, FHR93a, FHR93b]. It is currently being implemented in Smalltalk-80, but the concepts are independent of the implementation language.

Linguistic mechanisms behind this implementation provide further support for the notions of reuse, maintainability and incremental development in distributed transactional systems. These were introduced in an earlier version of Hermes/ST [FHR94], and centre on the specification of method and object properties through parameters

at method invocation and instance creation.

Following completion of the implementation, it is planned to test the system by coding various distributed applications in it. This process may well lead to a refinement of the default semantics provided, as well as the degree to which aspects of the base mechanisms are exposed to the framework.

I wish to acknowledge the assistance of my colleagues Michael Fazzolare and Bernhard G. Humm in this research.

References

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[BK91] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[CD89] Raymond C. Chen and Partha Dasgupta. Linking consistency with object/thread semantics: An approach to robust computation. In *Proceedings of the Ninth International Conference on Distributed Computing Systems (IC-DCS)*, June 1989.

[Dav73] Charles T. Davies. Recovery semantics for a DB/DC system. In *Proc. ACM 73 Nat. Conf.*, pages 136–141, Atlanta GA (USA), August 1973. ACM.

[EGLT76] K. P. Eswaran, J. N Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[EME91] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector (Eds.). *Camelot and Avalon.* Morgan Kaufmann Publishers, Inc., San Mateo, CA 94403, 1991.

[FAC+94] D. Finkelstein, D. Acton, T. Coatta, N. Hutchinson, and G. Neufeld. Object properties in the raven system. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, Poznan, Poland, June 1994.

[FHR93a] Michael Fazzolare, Bernhard G. Humm, and R. David Ranson. Advanced transaction semantics for TINA. In *Proceedings of the Fourth Telecommunications Information Networking Architecture Workshop (TINA 93), Volume 2*, pages 47–57, L'Aquila, Italy, September 27-30 1993.

[FHR93b] Michael Fazzolare, Bernhard G. Humm, and R. David Ranson. Hermes/ST user manual and technical manual. Technical Report No. 4,

Telecommunications Software Research Centre, Department of Computer Science, University of Wollongong, Wollongong NSW 2500, Australia, 1993.

[FHR94] Michael Fazzolare, Bernhard G. Humm, and R. David Ranson. Object-oriented extendibility in Hermes/ST, a transactional distributed programming environment. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming, Lecture Notes in Computer Science 791*, pages 240–261. Springer-Verlag, 1994.

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

[Gra78] J. N. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmueller, editors, *Lecture Notes in Computer Science, vol 60: Operating Systems: An Advanced Course*, pages 393–481. Springer Verlag, New York, 1978.

[HKM+94] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. Tinkertoy transactions. Technical report, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213-3891, 1994. Submitted to Conference on Lisp and Functional Programming '94.

[HMSC88] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery management in quicksilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[Hum93] Bernhard G. Humm. An extended scheduling mechanism for nested transactions. In Luis-Felipe Cabrera and Norman Hutchinson, editors, *Proceedings of the Third International Workshop on Object-Orientation in Operating Systems (IWOOOS'93)*, pages 125–134, Asheville, North Carolina, USA, December 1993. IEEE, IEEE Computer Society Press, Los Alamitos, CA.

[Lis84] Barbara Liskov. Overview of the Argus language and system. Programming methodology group memo 40, MIT Laboratory for Computer Science, February 1984.

[Lis88] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Mos85] J. Eliot B. Moss. *Nested Transactions – An Approach to Reliable Distributed Computing.* MIT Series in Information Systems. The MIT Press, Cambridge, Massachusetts and London, England, 1985.