Concurrent Transactions and Communicators: Extensible Synchronization Mechanisms for Distributed Programming

Ken Wakita and Yoshiki Ohshima
Department of Mathematical and Computing Sciences
Tokyo Institute of Technology
{wakita,ohshima}@is.titech.ac.jp

Abstract

Object-Oriented concurrency model has been recognized as good programming paradigm in concurrent and distributed programming. However, concurrency and synchronization primitives provided by the concurrent object-oriented languages are not expressive enough for complicated communication and synchronization schemes that are required by practical applications. The article proposes a highly extensible computation model that introduces two novel ideas of in concurrent object-oriented computation model: namely transactions and communicators. The concurrent transaction mechanism guarantees collective data integrity of a subcomputation and makes its activity appear atomic relative to the rest of the overall concurrent computation. The communicator is a programmable abstraction of message-based communication protocols. Communicator facilities allow the programmer to add arbitrary communication protocol to the programming language in a seamless manner with respect to the built-in communication primitives. Moreover, by combined use of concurrent transaction and communicators, various atomic communication primitives can be easily developed. A number of examples demonstrate the expressiveness given through these facilities in describing various distributed systems.

1 Introduction

Concurrent object-oriented computing has drawn much attention for concurrent and distributed programming [18, 17, 4]. A number of concurrent object-oriented languages (COOLs) have been designed and implemented on various platforms, including shared memory parallel computers, network connected clusters of workstations, and massively parallel machines. COOLs hide the underlying hardware architectures from the programmer's point of view and provide convenient tools for expressing concurrency, communication, and synchronization. Past research and practice on COOLs have shown their effectiveness in development of highly concurrent distributed applications[18, 1].

However, programming large distributed systems still remains extremely difficult. An obstacle is COOLs' lack of abstraction mechanism for controlling complexity in distributed systems. In sequential programming, one of the complexity of the system is manipulation of various data structures. To deal with the wide variety of data structures required by applications, sequential programming languages provide data abstraction mechanisms such as user-defined data types, abstract data types, classes, modules, etc. These facilities enable a modular way of designing and constructing a system a larger and complex system from smaller and less complicated subsystems.

Though data abstraction is also important in distributed programming, they do not solve sort of complexities that are peculiar to distributed computing such as concurrency, communication, and synchronization. Distributed systems require various communication protocols and synchronization schemes that differ from systems to systems. However, most distributed programming languages including COOLs provide useful but a fixed set of primitives, whose abstraction level is often much lower than that of communication and synchronization schemes required by the distributed application. As the result, the pro-

grammer are forced to construct complex communication and synchronization schemes sacrificing modularity of the code.

The COOL proposed in the article, HARMONY/2, tries to solve the above mentioned problem by providing two abstraction mechanisms: namely *concurrent transactions*, a powerful primitive for specification of global concurrency control, and *communicators*, an extensible communication mechanism.

By using transactions, the programmer can easily describe many classical but still difficult problems in distributed programming, which otherwise are extremely difficult to describe with current COOLs. Examples of such problems are deadlock-free systems, controlling concurrent accesses to shared data, atomic execution of concurrent activities, termination detection of distributed processes, check pointing, and rollback. To integrate transaction semantics [3, 2] with concurrent and asynchronous nature of COOLs, HARMONY/2 incorporates a new transaction model, which we call *concurrent transaction model*. Concurrent transaction model is an extension of traditional nested transaction model that allows each transaction to involve concurrency, however still preserving characteristics of traditional transaction models such as atomicity, isolation, and serializability.

The communicator is an abstraction mechanism of message-based communication. It allows the programmer to define user-defined communication primitives and use them seamlessly with the built-in communication primitives (just like, user-defined data types and basic data types are seamlessly used in sequential programming languages). The examples of user-defined communication primitives include asynchronous communication, multicasting, future type message passing, message monitor, message encryption, and concurrent aggregates [13, 14].

Concurrent transactions and communicators are diagonal notions in HARMONY/2. Hence, it is possible to use user-defined communication primitives within a concurrent transaction. Because communicator mechanism is realized in terms of HARMONY/2 objects that are under domination of the transaction manager, concurrent transaction mechanism properly maintains transaction semantics over the behavior of user-defined communication primitives. It is also possible to use concurrent transactions within a definition of a communicator. This allows the programmer to make *user-defined atomic communication primitives*, such as atomic RPC, atomic future, etc. Examples show that this diagonal design of concurrent transactions and communicators are effective in describing many important distributed algorithms.

The article proposes a concurrency model that unifies traditional concurrent object-oriented computation model with concurrent transactions and communicators, introduces the language HARMONY/2 whose design is based on the computation model, and illustrates its expressiveness. The structure of the rest of the article is as follows. Section 2 explains the basic feature of a COOL HARMONY/2. Section 3 and section 4 introduce the concept of nested concurrent transactions and first class messages, respectively and Section 5 shows examples. Section 6 compares the work with previous research and section 7 summarizes the article.

2 HARMONY/2

HARMONY/2 is a class-based COOL. The following is a simple example of class definitions:

```
(Class Container (v)
  (Initial (val) (v := val))
  (Method (assign val) (v := val))
  (Method (refer) (Reply v)))
```

It defines a class named Container that has an instance variable named v. The **Initial** clause defines an initialization method of the class. When an instance of this class is created with a **New** form, for instance:

```
(New Container 0)
```

a new instance of Container class is created and the initialization method is executed automatically on the new instance. In the above example, the argument given to **New** form, 0, is passed to the formal

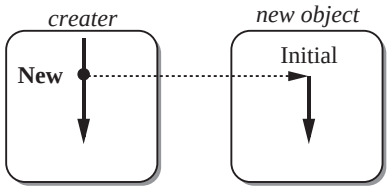

Figure 1: Object creation

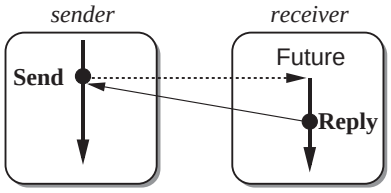

Figure 2: Send-and-Reply style communication

argument of the initialization method and is assigned to the instance variable v. A class definition may contain one or more method definitions. Container class has two, named assign and refer. The method is invoked in response to message acceptance. The assign method takes one argument (val), and assigns it to v. The refer method replies the content of v to the sender of the message. For instance, the following expression,

```
(let ((p (New Container 5)))
  (Send p refer))
```

creates an instance of Container named p and then sends it a refer message. The result of the message passing is the reply value of the refer method, namely 5.

An object is in one of three execution states: namely *dormant*, *active*, or *waiting*. If a method is not executing any method, it is dormant. A dormant object accepts a message and becomes active. An active object can perform (1) reference and assignment to the instance variables, (2) creation of new objects (**New**), (3) sending messages (**Send**), and (4) sending reply (**Reply**). Execution of **New** form (object creation) immediately results in the pointer to the new object and thus the creator proceeds method execution. The initialization method is executed on the new object asynchronously relative to method execution of the creator (see Figure 1). When an active object sends a message, the object becomes waiting. A waiting object does nothing but waits for the reply . When it receives one it becomes active again and continues the rest of method execution (Figure 2). All these mean that an object is single-threaded and each object's processing of messages do not overlap (absence of intra-object concurrency).

There are two ways to increase the degree of concurrency: namely, *early reply* and object creation. Figure 2 depicts the send-and-reply style communication scheme of HARMONY/2. After the receiver replies the sender back, both the sender and the receiver proceeds their method execution. As depicted in Figure 1, method of the creator object and initialization method of the new object are executed concurrently.

3 Concurrent Transactions: Transactions for COOLs

This section gives an brief overview of concurrent transaction mechanism. This mechanism has been incorporated in HARMONY/1, which is the direct ancestor of HARMONY/2. For more detail, the reader is referred to [15]. To motivate the design, let's consider the following example.

Suppose there are two instances, o_1 and o_2, of Container class and one wants to reset their values to zero *at the same time*: in other words, without possible interference from other concurrent activities.

Though object-wise single-threadedness guarantees execution of each assign method to be atomic, simply sending two assign message does not suffice because collective atomicity is not guaranteed for multiple message passing: in presence of concurrency, it is possible for other concurrent activities to intervene processing of two messages.

This is a simple but typical problem in concurrent programming. With traditional COOLs, a solution is to add a lock mechanism to Container. To deal with possible deadlocks, the programmer may also need to implement deadlock avoidance protocol or deadlock detection and recovery mechanism. There are three major drawbacks of this approach. Firstly, the modularity of the code is significantly reduced. Though the original definition of Container is straightforward, the additional lock management code complicates the class definition and reduces its understandability and maintainability. Secondly, avoidance, detection, and recovery of deadlock states are the most difficult distributed algorithm to implement. Lack of proper synchronization mechanism that can express the above problem will leave most programmers out of distributed programming. Lastly, reusability of the code is significantly reduces. In the above approach, Container class with lock enhancement is so specialized to the configuration and deadlock avoidance policy, it is not possible to reuse the implementation of Container class.

The notion of transactions has been incorporated in HARMONY/2 to express global synchronization problems such as the above mentioned one. A *transaction* in HARMONY/2 is an atomic execution of part of the concurrent computation called *subcomputation*, which will be explained in the next paragraph. A transaction is specified by a transaction block in the method definition. For instance, a solution to the above mentioned problem can be described as follows[1]:

```
(Method AtomicUpdate (o1 o2)
  (Transaction (Send o1 assign 0) (Send o2 assign 0) (Send o1 refer)))
```

A transaction block specifies the subcomputation that is a *collective effect of executing the code within the transaction block*. For instance, in the above example, the corresponding subcomputation involves sending three messages to o_1 and o_2, processing three messages by o_1 and o_2, and receiving reply from o_1 for processing the refer message. In general, the subcomputation specified by a transaction block includes execution of the transaction block (update of the object, object creation, and message passing). It also includes computation initiated by execution of the transaction block: execution of initialization methods on the objects created by the transaction block and processing of the messages created by the transaction block. Moreover, the subcomputation includes method executions initiated by the method executions which in turn are initiated by the execution of transaction block, and so on.

Concurrent transaction mechanism makes a subcomputation specified by a transaction block to be executed as an atomic transaction in the traditional sense: concurrent transaction mechanism guarantees three properties of the transaction, namely *atomicity*, *isolation*, and *serializability* [3, 2]. Atomicity is an all-or-nothing semantics: a transaction either succeeds completely or fails and leaves no effect. Success of a transaction is called *commitment* and failure is called *abortion*. In the above example, the three messages are either processed completely in case of commitment or unprocessed completely leaving no effect of message sendings as if the message passing has never been performed in case of abortion. Isolation property guarantees that partial execution of the transaction is invisible to other concurrent activities. Even though there may be concurrent activities that try to get the values of o_1 and o_2, isolation property guarantees that those activities do not observe the partial execution of the transaction. For instance, the concurrent activity does not observe the partially updated state of the transaction where o_1 is reset to zero while o_2 is not[2]. Serializability gives the transaction a view of the system where the system is exclusively accessed by the transaction. For instance in the above example, even though a concurrent activity is trying to update o_1, it cannot intervene execution of the transaction and thus the refer method always returns zero.

[1] The third message passing is not required but it is there to address more about concurrent transactions.

[2] Note that like nested transaction, nested concurrent transaction relaxes serializability and isolation to allow descendent transaction to be partial atomic execution of the ancestor transaction. Note also that in presence of concurrency inside the transactions, execution of the subtransaction is atomic relative to both its sibling subtransactions and its parent transaction.

Like traditional nested transaction [12, 11], a concurrent transaction may arbitrarily nested: a concurrent transaction may spawn a subtransaction. A significant difference is that due to asynchrony in the computation model a concurrent transaction involves multiple concurrent activities while with the nested transaction model, a transaction is a sequential execution of the code that has no concurrency. Therefore, it is the case with the concurrent transaction model that a concurrent activity in a transaction T spawns a subtransaction T_s that is executed concurrently with other concurrent activities in T. In the nested transaction model, due to the sequential nature of the transaction, there is no concurrency between a transaction and its subtransactions.

To guarantee atomicity, serializability, and isolation properties of the subtransaction relative to its parent transaction in concurrent transaction model, the transaction manager controls concurrency between a parent transaction and its subtransaction as well as between sibling transactions. An interesting point is that though execution of subtransaction appears atomic relative to its parent and sibling transactions, execution of the parent transaction appears as non-transactional relative to its subtransactions. This concurrency control can be achieved by extending lock compatibility rules of the traditional nested transaction mechanism.

Another difference is completion semantics of transactions. In the concurrent transaction model, completion of execution of a transaction block does not necessarily mean the collective completion of the subcomputation of the transaction. For instance, commitment of the transaction should be postponed until the termination of the transaction. This mechanism is implemented by the object scheduler of HARMONY/2. Details of the implementation of concurrent transaction mechanism is out of the scope of this article. They appear in our forthcoming paper together with the formally defined operational semantics of the concurrent transaction.

HARMONY/2 supports two different behavior for abortion of transactions: *repeat-until-commitment* and *abort*. **Transaction** declaration specifies the former. The transaction manager repeatedly executes the transaction after each abortion until it commits. The latter behavior is specified by **Transaction***$*$ block. In this case, the transaction executes once and it either commits or aborts. The execution status can be tested with $if\_committed$ form, which is used in the following manner:

```
(if_committed (Transaction* statement ...)
    true_case
    false_case)
```

4 Communicators: Extensible Communication Construct

The extensible communication mechanism of HARMONY/2 has been designed from the observation that in spite of the variety of communication protocols required by distributed applications, COOLs provided fixed set of communication primitives. HARMONY/2 provides a simple but highly extensible mechanism called *communicator* that can express various communication protocols including multicasting protocols, future-type message passing, concurrent aggregates, message monitors, etc. Theoretical background and many examples are found in [13, 14].

A notable point of communicator mechanism is that underlying implementation issues of HARMONY/2 is completely hidden from the programmer's view. Hence, the programmer only needs to know about the concurrency model of HARMONY/2 to implement a communicator: knowledge about implementation issues such as the software architecture of the compiler and the runtime system, the operating system support, and the underlying hardware architecture is not required.

A communicator is defined by **Communicator** form. For instance,

```
(Communicator Async (msg)
  (Send msg reply nil)
  (Send msg fire))
```

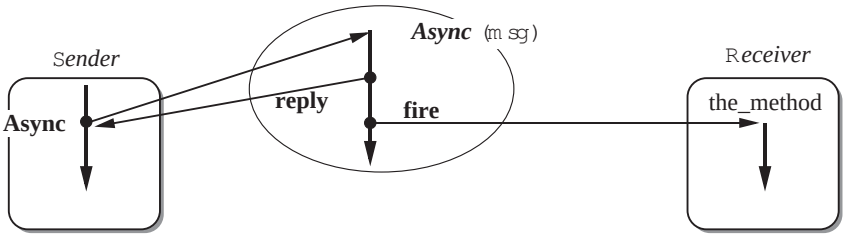

Figure 3: Implementation scheme of **Async** communication protocol

defines a communicator named **Async** that implements the behavior of asynchronous message passing. It also introduces to HARMONY/2 following new syntactic form for expressing asynchronous message passing.

(**Async** *receiver* the_method *argument*)

When this message passing form is executed, the message is manipulated as defined by **Async** communicator. The argument given to the **Communicator** declaration, msg, is a metalevel representation of the message: it contains such information as the sender, the receiver, the method selector, and the message of the message. We call message representations *first class messages* because they are in fact instances of a HARMONY/2's built-in class named Message. First class messages are manipulated through methods provided by Message class: namely copy, fire, and reply. Copy duplicates the first class message, fire converts a first class representation of the message into active message that is delivered to its target, and reply sends reply to the object that is specified as the reply destination of the first class message. For instance, **Async** communicator, performs reply operation and then fire operation on the first class message, msg. The reply let the sender receive the result (nil) for message sending in **Async**-mode. At this point, the sender continues its method execution and the rest of the message manipulation by the communicator is performed concurrently relative to the sender. The fire operation converts the the first class message (msg) into an active message, which is delivered to and processed by the target ($receiver$). This asynchronous communication scheme is depicted in figure 3.

We can define various *atomic communication protocol* by using transaction blocks in communicator declarations the first class message within a transaction block. The following example realizes an atomic message passing protocol similar to atomic remote procedure call of Argus [11]:

(**Communicator AtomicSend** (*msg*)
  (**Transaction** (**Send** *msg* fire)))

AtomicSend simply performs fire operation on the first class message within the transaction block. The surrounding transaction block makes the collective effect of fireing atomic: message sending, message processing, and its transitive effect.

This is a very modular approach in introducing atomic communication protocols to the programming language because the programmer needs only to replace **Send** in the message passing form by **AtomicSend** to make the message passing atomic, in the following way.

(**AtomicSend** *the_receiver* the_method *argument*)

5 Examples

This section gives four programming examples of transaction facilities of HARMONY/2. The mechanism of transaction management consists of various useful functionalities. This section shows that various runtime mechanisms of concurrent transaction facilities are useful in implementing distributed algorithms and also that these mechanisms are uniformly expressed by concurrent transactions. Subsec-

tion 5.1 presents a deadlock-free implementation of the dining philosopher problem using deadlock manipulation mechanisms such as inter-transactional deadlock detection and rollback mechanisms. In subsection 5.2, completeness property of the transaction serves for checkpointing global states of the objects and explicit abortion primitive serves as rollbacking. Subsection 5.3 shows isolation property of transaction can be used to describe global process termination. Lastly, subsection 5.4 describes selective commitment of parallel processes can be realized by explicit abortion of transactions.

5.1 Deadlock-free System (Dining Uncooperative Philosophers)

The problem of dealing with deadlock states is a classical but still difficult problem to express with most concurrent programming languages. As the result, the programmer has to construct a deadlock avoidance protocol that complicates the code and significantly reduces its modularity. We show through dining philosopher problem that synchronization specification using nested concurrent transactions is very effective for this type of problems. It implements a *protocol-less implementation* of dining philosophers problem: each philosopher is independent from each other and he/she picks up forks *without paying attention to other philosophers' behavior*. In spite of the protocol-less implementation, the code is free of deadlock because the critical region, where the philosopher picks up forks, is made atomic by a transaction block. The transaction manager resolves possible deadlocks. If transactions run into a deadlock state, the transaction manager aborts some of them and let others proceed. Otherwise the transaction eventually commits, which means a philosopher successfully picks up forks and enjoys the meal.

```
(Class Philosopher ()
  (Initial () (Async self (Think)))
  (Method (Think)
    (think_over_and_over) (Async self Eat))
  (Method (Eat)
    (let ((fk1 (choose_left_or_right)) (fk2 (another)))
      (Transaction (Send fk1 grab) (Send fk2 grab))
      (enjoy_your_meal)
      (Send fk1 release) (Send fk2 release)
      (Send self Think))))
```

5.2 Checkpoint and Rollback (Hybrid Algorithm)

When we face a complex problem such as NP problems, it is often the case that there is an efficient and effective algorithm that is applicable to many cases but not to general cases. In such situation, it would be good to try the efficient one first and if it fails then try the inefficient but generic algorithm next. The following is the skeleton of such hybrid approach. Within the transaction it tries the efficient (but incomplete) algorithm. After a while, if the efficient algorithm is known to fail then the the efficient algorithm aborts all its side-effects and tries the generic (but inefficient) algorithm.

```
(if_committed (Transaction* (incomplete_algorithm))
  'ok
  (generic_algorithm))

(define (incomplete_algorithm)
  (try_the_incomplete_algorithm)
  (if (impossible_to_continue) (Abort))
  (rest_of_the_job))
```

The transaction block surrounding the invocation of the incomplete algorithm can be considered as a global snapshot of the initial state of the system when the algorithm starts. Explicit abortion by **Abort** primitive terminates the transaction and reverts all the updates made by the incomplete algorithm. Another purpose of the transaction block is *termination detection.* In presence of concurrency it is possible

that even after the algorithm produces the result, some part of the concurrent algorithm is still continuing some computation. It is desirable that the complete termination of the algorithm can be detected. Atomicity property of nested concurrent transactions guarantees that once the transaction commits, the entire computation of the transaction has completely finished.

5.3 Termination Detection and Concurrency Control (Atomic AND-parallelism)

There is a class of problems that can be decomposed into several mostly independent subproblems. These types of problems have been extensively studied in parallel logic programming languages. Parallelism obtained by running these subproblems is called *AND-parallelism*. Though we can naturally retrieve parallelism from this type of problems, avoidance of interference between computation of the subproblems generally requires complicated concurrency control. The following solution makes each subproblems to be computed as a transaction.

A notable point is that it implements an atomic version of *future mode* communication scheme [7, 17, 16]. When a sender sends a message in future mode, the sender immediately receives a place holder of the reply called *future object*. Also the message is sent to the receiver that eventually stores the reply in the future object. As the result, both the sender and the receiver runs concurrently in a similar manner as in asynchronous message passing. The sender may communicate with future object to get the result at arbitrary point of time. At that moment, if the reply is available at the future object the reply is sent to the sender; otherwise the sender synchronizes with the reply from the receiver. The following implementation of atomic future protocol creates an instance of Future class and sends it to the sender as the reply. The future object activates the message on behalf of the sender (fire). The surrounding transaction block makes the message passing, method invocation (including its collective effects), and receiving reply atomic. The sender obtaines the reply by sending value message to the future object.

The body of the algorithm is expressed using **AtomicFuture** communicator. The atomic future message sendings are done in parallel because message activation is performed by the future object instead of the sender. Finally, the sender waits until each transaction commits and the result of each subproblem becomes available at the corresponding future object.

```
(Class AFuture (result)
  (Initial (msg) (result := (Transaction (Send msg fire))))
  (Method (value) (Reply result)))

(Communicator AtomicFuture (msg)
  (let ((msg1 (Send msg copy)))
    (Send msg reply (New AFuture msg1))))

(let ((f1 (AtomicFuture you do_this))
      (f2 (AtomicFuture him do_that))
      (f3 (AtomicFuture her check_it)))
  (collect_result (Send f1 value) (Send f2 value) (Send f3 value))))
```

5.4 Selective Commitment (Atomic OR-parallelism)

Suppose processing a query that tries to find, in a tree of objects, an object with a matching key value. We gain parallelism by dividing the tree into several subtrees and executing concurrent subqueries on them in a concurrent manner. If the inquirer is satisfied by an answer from a subquery, it is desirable from efficiency that immediately after some process finds the answer all other processes stop their inquiry. The following example defines a new communicator called **MultiFuture**, which, like **AtomicFuture**, activates the message asynchronously and stores the reply in a Container object. Unlike **AtomicFuture**, every **MultiFuture** sending does not store the result in distinct future objects but instead stores in the same container object. Finally, **AtomicFuture** forces all the other subtransactions to stop and refresh all their side-effects with **Abort_Siblingtransactions**.

```
(define p (New Container))

(Communicator MultiFuture (msg)
  (Transaction (Send msg reply p) (Async p assign (Send msg fire))
      (Abort_Siblingtransactions)))

(begin
  (Transaction (MultiFuture left find key) (MultiFuture middle find key)
               (MultiFuture right find key))
  (Send p value))
```

6 Related Work

Moss's nested transaction: There are several distributed programming languages that support transactions. Argus [11] is the first among them that were designed and implemented. Argus is a distributed programming language that introduced an abstraction of concurrent computing agents called *guardians*. Guardians can be considered multi-threaded objects that communicate each other by remote procedure calls (RPC). Concurrency inside the guardian is controlled by atomic semantics given to each remote procedure call. Argus treats each remote procedure call as a transaction to provide the programmer with transparency against possible hardware problems, such as network failures and media failures.

Nested transaction model developed by J.E.B. Moss is a theoretical foundation of Argus [12]. Unlike traditional transaction mechanisms, Moss's transaction can spawn another atomic (sub)transaction and make its execution as a part of the former. Advantages to the traditional models are:

- Localization of partial failures: Failure of a subtransaction does not lead to the failure of its parent. Instead the parent can issue commitment if the next trial of the subtransaction succeeds. This localization of atomic transactions substantially reduces the cost for abortion.
- Increased concurrency: Though nested transaction model assumes sequentiality of transactions, multicasting RPCs spawn multiple subtransactions that can be executed concurrently.
- Ease in programming: Due to concurrent processing of multiple requests at the same guardian, a guardian is inherently a concurrent object. However, all the requests being processed atomically, the programmer need not worry about controlling concurrency between them.

Other distributed programming languages that has built-in nested transaction facilities include Avalon/C++[5] and Aeolus [10].

Unfortunately, the *sequentiality assumption* imposed by Moss's nested transaction mechanism is too restrictive, when we incorporate transaction mechanism in COOLs.

Concurrent transactions: HARMONY/1, which is the direct ancestor of HARMONY/2, is a COOL that incorporated an extended nested transaction mechanism, what we call *concurrent transactions* [15]. Concurrent transaction model relaxes the sequentiality assumption of the nested transaction model and allows multiple methods to be executed concurrently in a transaction. HARMONY/1 separated message passing semantics and transaction semantics: in HARMONY/1, the programmer can specify part of the program, where transaction semantics is required, to be executed as a transaction and keep the rest non-transactional so that it is executed without extra overhead for transaction management. Another benefit of this feature is that the reduced number of created transactions because each message passing does not necessarily create a transaction. Hence, costs required for transaction management, such as creation, commitment, abortion, rollbacking can be minimized. A similar mechanism has been incorporated in Actor-like distributed programming language called Hermes/ST [8] and a concurrent SML system in Venari project [6].

HARMONY/2 also incorporates concurrent transactions. The article focuses on the expressiveness of concurrent transactions facilities, especially when combined with the extensible communication mechanism.

Multiple concurrency control policies: Kaiser and others, stressed the importance of supporting multiple concurrency control policies by the distributed programming language [9]. MELD is a COOL that integrates two primitives of concurrency control: *atomic block* as an computationally efficient but lower level primitive and *atomic transaction* as more expressive mechanism. Through many examples, [9] shows how effective a concurrent programming language is to have multiple concurrency policy mechanisms.

The communicator mechanism of HARMONY/2 has been designed in the same spirit as MELD but has achieved much more flexibility through its extensible communication features. The programmer can define various concurrency and synchronization schemes in terms of communicators and add them to HARMONY/2. Also the concurrent transaction mechanism and communicator mechanism are integrated so that the programmer can define various *atomic* communication protocols. In this sense, HARMONY/2 can be viewed as a concurrent programming language with *arbitrarily many concurrency control policies*.

7 Summary

The work introduced two concepts, concurrent transactions and communicators, in a concurrent object-oriented programming language, HARMONY/2. Concurrent transaction facility is a powerful mechanism for expressing global concurrency control. Communicator facility is an abstraction mechanism for adding user-defined communication primitives to HARMONY/2.

One of the contributions of the work is to show that transactions are useful not only for concurrency control over database applications such as bank account example [11, 6] but also for description of various distributed algorithms that were extremely difficult to describe with traditional distributed/concurrent programming languages, including COOLs. Another contribution is to show that an extensible communication mechanism and transaction mechanism can inhabit symbiotically in one programming system. More importantly, both features can be combined to make various user-defined atomic communication primitives, such as atomic RPC and atomic future mode message passing.

Ongoing research activities include formal work and implementation. As for formal work, we have defined an operational semantics of HARMONY/2 based on lambda value calculus and transition system that captures both concurrent transactions and communicators. This means that we have formally expressed the integrated semantics of object-oriented concurrent model and transaction-based concurrency control mechanism. Based on the operational semantics, we are going to formally prove that concurrent transactions preserve traditional transaction semantics, namely atomicity, isolation, and serializability. This work will appear in our forthcoming paper. As for implementation work, concurrent transactions have been incorporated in a COOL HARMONY/1, which is a direct ancestor of HARMONY/2, and tested with many examples [15]. Communicator mechanism has been implemented in HARMONY/2 and used for description of various communication mechanisms[]. Incorporation of concurrent transaction facilities in HARMONY/2 is being completed.

Acknowledgment Thanks go to people whom we met at monthly workshop on transaction technology. Shigekazu Inohara, Yasuro Kawata, Masataka Sassa, Etsuya Shibayama, and Keitaro Uehara read early drafts of the article. The authors are grateful for support and encouragement by colleagues at our department.

References

[1] G. Agha, P. Wegner, and A Yonezawa, editors. *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[2] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2), 1981.

[3] S. Ceri and G. Pelagatti. *Distributed Databases: Principles & Systems*. McGrawHill, New York, 1985.

[4] A. Chien. *Concurrent Aggregates*. MIT Press, 1992.

[5] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, 21(12):57–69, 1988.

[6] N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *Transactions on Programming Languages and Systems*, 16(6):1719–1736, 1994.

[7] R. H. Halstead, JR. Multilisp: A language for concurrent symbolic computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[8] B. G. Humm. An extended scheduling mechanism for nested transactions. In *Proceedings of the third International Workshop on Object-Orientation in Operating Systems*, pages 125–134, December 1993.

[9] G. E. Kaiser, W. Hseush, S. S. Popovich, and S. F. Wu. *Multiple Concurrency Control Policies in an Object-Oriented Programming System*, chapter 7, pages 195–211. MIT Press, 1993.

[10] R. J. LeBlanc and C. T. Wilkes. Systems programming with objects and actions. In *The 7th International Conference on Distributed Computing Systems*, pages 132–138. IEEE, IEEE Computer Society Press, 1985.

[11] B. Liskov. Distributed programming in Argus. *Commun. ACM*, 31(3):300–312, 1988.

[12] J. E. B. Moss. *Nested Transaction: An Approach to Reliable Distributed Computing*. The MIT Press, Cambridge, Massachusetts, 1985.

[13] K. Wakita. First class messages as first class continuations. In *proceedings of International Symposium on Object Technologies for Advanced Software (ISOTAS '93), LNCS 742*, pages 442–459, Kanazawa, November 1993.

[14] K. Wakita. First class continuation facilities in concurrent programming language Harmony/2. In *proceedings of International Workshop on Theory and Practice of Parallel Programming (TPPP '94), LNCS 907*, pages 300–319, Sendai, November 1994.

[15] K. Wakita and A. Yonezawa. Linguistic supports for development of distributed organizational information systems in object-oriented concurrent computation frameworks. In *ACM Conference on Organizational Computing Systems*, pages 185–198, November 1990.

[16] Y. Yokote and M. Tokoro. The design and implementation of ConcurrentSmalltalk. In *Object-Oriented Programming Systems, Languages and Applications*, volume 21(11), pages 331 – 340. SIGPLAN Notices (ACM), November 1986.

[17] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. MIT Press, Cambridge, Mass., 1990.

[18] A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming*. The MIT Press, Cambridge, Massachusetts, 1987.