First Class Messages as First Class Continuations*

Ken Wakita

Tokyo Institute of Technology
Department of Information Science
2-12-1, Ookayama, Meguro-ku
Tokyo 152 Japan
e-mail: wakita@is.titech.ac.jp

Key Words: Object-oriented concurrent programming, first class message, first class continuation, extensibility, communication, synchronization

Abstract

First class messages, which we call message continuations, provide object-oriented concurrent programming languages with extensibility in modeling and programming communication schemes such as asynchronous communication, multicasting, sophisticated synchronization constraints, inter-object synchronization, concurrency control, resource management, and so on. In spite of its powerful extensibility, the framework is sound in that the framework guarantees that no program can destroy the semantics of the built-in communication primitives. This good property was obtained by categorization of message continuations and careful design of the primitive operations on message continuations.

1 Introduction

Object-oriented programming languages provides powerful mechanisms for data abstraction and code reuse through encapsulation, inheritance, delegation, polymorphic type substitution, etc. However most object-oriented programming languages have very limited support of modeling and describing control structures (or communication structures). Due to this lack of extensibility in the dimension of *control abstraction*, the programmer often suffers from contaminating method definitions with fractions of code that implement protocols of control/communication structures. For instance, when asynchronous communication is required, the programmer has to modify the code of both the sender and the receiver to add asynchronous protocol. A flaw here is that the code that implements communication protocol is scattered over all the participants of asynchronous communication, leading to degradation of modularity and poor maintainability of the program. Another and still more serious problem is that addition of asynchronous protocol makes the reuse of classes more difficult because the method code includes part of the protocol, which was not part of the specification of the original class. Needless to say, concurrent programming languages should support such a simple communication primitive as asynchronous communication. In practice, many communication primitives, including asynchronous message passing, have been proposed and incorporated in concurrent languages. However, the point is that even the most sophisticated set of communication protocols can not deal with all different kinds of communication schemes required by wide variety of applications. When a new communication scheme is required, lack of extensibility will exhibit the similar problems as those we found in adding asynchronous communication will take place. The goal of our research is not to propose yet another communication primitive, but to give powerful extensibility to concurrent languages, whereby almost any kind of communication scheme is programmable.

Lack of extensibility in modeling communication becomes a serious problem in distributed programming, because each application program requires different forms of communication and different synchronization schemes. Existing object-oriented concurrent programming languages have various synchronous/asynchronous communication primitives [19, 20, 3, 13, 10, 6] and some languages have

*The work will be presented at International Symposium on Object Technologies for Advanced Software (ISOTAS) built-in concurrency control features [5, 16]. However, these powerful but non-extensible communication primitives are not sufficient for modeling and describing more advanced communication/synchronization schemes, such as flexible concurrency control [5, 16], notification [8], intelligent negotiation platform [7], etc.

The article introduces the concept of *first class continuation* into concurrent computation models. A *continuation* of a expression is the *rest of the computation* relative to the evaluation of the expression. Continuations can be defined for sequential execution of any program. The notion of continuations was introduced in describing denotational semantics of control structures such as **goto** statement and **while** loop [15]. To the author's knowledge, Scheme [14], a dialect of Lisp, is the first programming language that incorporated first class continuations as built-in language feature. Scheme provides a primitive function which captures the current continuation as a form of a function. First class continuations, thus obtained, can be bound to a variables and stored in data structures. A first class continuation is represented as a function, sometimes called an *escape function*. Invocation of a first class continuation results in giving the control back to the time when the continuation was obtained. Though Scheme has very small number of built-in control structures, it is known that more complicated control structures such as *catch-and-throw* and *unwind-protect* can be implemented using first class continuations. Furthermore, continuations allow us to express more advanced features such as non-local jumps, exception handling, backtracking, and co-routines [9, 17].

We introduce the notion of first class continuations into object-based concurrent computation. Computation in object-based concurrent computation models has two dimensions of computation, namely *internal computation* (execution of methods) and *external computation* (communication between objects via messages). Because of this duality in object-based computation, we can define two independent classes of continuations, namely **method continuation** and **message continuation**. A *method continuation* is "the rest of the computation" with respect to execution of a method. From similarity between method execution and function evaluation, method continuations can be considered as an object-based version of ordinary continuations. A *message continuation* is a message itself. A message determines the rest of the computation in that it determines which method to invoke at which object (the *target*), to which object to send the reply (the *reply destination*), etc. Thus a message is a good device for expressing a continuation with respect to external computation. We have designed an object-based concurrent language, called HARMONY/2, which integrates both dimensions of continuations with object-based computation models. The article focuses on the message continuation features[1]. First class message continuations allow us to describe various communication schemes such as synchronization schemes, message scheduling policies, emulation of multi-processor environment, concurrency control, system monitoring, notification mechanism, distributed constraint resolving, and so on.

Particular emphasis is on **soundness** of the framework. One of the problem with highly extensible systems is that programmer's customization may destroy the semantics of the built-in primitives of the system, thus making it impossible to reason about the behavior of the system. To protect the system from dangerous customization, most extensible systems impose certain disciplined programming style on the programmer. The system guarantees system's correct behavior as far as the programmer do not violate the discipline.

In designing the message continuation framework, we took more conservative and safer approach. The system, rather than the programmer, is responsible for protecting system's semantics. Thus the framework imposes no discipline but still the semantics of the built-in primitives never changes, whatever code is executed. We claim that our framework is **sound** with respect to this safer property compared with most extensible systems. Soundness was obtained by careful design of primitive operations on message continuations. To achieve soundness, we introduced two categories of message continuations, namely **asynchronous message continuations** and **synchronous message continuations**.

In the rest of the article, the object-based computation is introduced in Section 2, the message continuation framework and its soundness is explained in Section 3, and Section 4 shows several programming examples with message continuations. Section 5 compares the framework with related work and Section 6 concludes the article.

[1] Uniform integration of method continuations and message continuations will be reported in the forth-coming article.

2 Computation Model

Message continuation framework presented in the article can be tailored to various message-based distributed computation models. However, to clarify the discussion, we use the syntax and semantics of a subset of HARMONY/2. HARMONY/2 is an object-based concurrent language which is designed and being implemented at Tokyo Institute of Technology. Its syntax is Standard ML [12] with extensions of object-oriented features. Inter-object communication is based on RPC-like message passing. The computation model is a multi-threaded model; an object may have more than one methods execute concurrently.

```
Class BinPlus
    var x = 0;   var y = 0;      ①
    initialize(a, b) = (x:=a;  y:=b)     ②
    mth result() = ↑(x+y)     ③
EndClass
```

The above is an example of class definition. It defines a class named BinPlus, which implements a binary addition operator. The class declares two instance variables (x and y; ①) and defines the initialization method (②) and a method that returns the sum of the instance variables (result; ③). The instance variable declaration using **var** has similar syntax as **val** declaration Standard ML. The value assigned to an instance variable is referred to by its name. Unlike values of Standard ML, an instance variable may be reassigned to another value by using assignment operator (:=). An **initialize** declaration defines an instance initialization method. When an instance is created, the initialization method is invoked. A **mth** declaration defines a method. The syntax is similar to **fun** declaration of Standard ML. The body of a method code may contain arbitrary expression of Standard ML. Additionally, following three special forms are allowed to appear in the method body:

Instance Creation Form:	Class ← **New**(a_1, a_2, . . .)
Message Passing Form:	object ← method_name(a_1, a_2, . . .)
Reply Form:	↑*value*

Special Forms of HARMONY/2

Execution of an *instance creation form* creates an instance of the specified class. In the example below, an instance of BinPlus class is created and the instance is bound to an instance variable named p. The parameters given to an instance creation form (e.g., (3, 5)) are passed to the initialization method which is declared by the **initialize** declaration. The second line of the example is message passing to the object referred to by p. Conceptually, a message is created and sent to p (the *target* of the message). The identifier following ← is called the *method selector*. When the message is delivered to the target the message invokes the method which has the same name as the method selector.

```
let var p = BinPlus ← New(3, 5)
in   p ← result()
end
```

The presented computation model is based on RPC-like synchronous message passing; the sender of the message waits for the receiver to send reply back. Though, HARMONY/2 supports both synchronous and asynchronous message passing, we did not add asynchronous one in the presented model as a built-in feature, trying to make the model simpler and avoid confusion between the built-in asynchronous primitive and asynchronous communication protocols implemented by message continuations.

The model is a multi-threaded model. Whether or not a message is being processed in an object, a delivered message is immediately accepted and corresponding method starts execution. Consequently, multiple methods may execute in an object. In other words, an object has internal concurrency. The programmer can use method guards to control internal concurrency.

As the model does not have asynchronous message passing nor multicasting protocol, the reader might think there is no source of concurrency in the model. Two sources of concurrency are early reply mechanism and instance creation. A method may send reply before it completes the execution. After sending reply, the sender and the receiver execute concurrently. Instance creation also increases the number of concurrency. Execution of an instance creation form produces a new instance, replies the object identifier back to the creator, and then invokes the initialization method on the new instance. Consequently, the rest of method execution at the sender side and execution of initialization at the new instance execute concurrently.

3 Message Continuations

This section presents the message continuation framework. A message continuation is a first class message obtained through **message conversion**. Three primitive operations are defined for message continuations. Subsection 3.1 gives an overview of the message continuation framework. Subsection 3.2 exemplifies the use of message continuations. Subsection 3.3 goes into further detail of the semantics of the primitive operations and explains **soundness** property of the framework.

3.1 Message Conversion

The principle idea of message conversion is to convert a message into a first class object so that the message becomes accessible in the method code. We call this process *message conversion* and the result of conversion, namely the first class message, a *message continuation*.

HARMONY/2 represents a message continuation as an instance of MsgCont class, which is one of the built-in classes of HARMONY/2. Conceptually, a message continuation contains all the information that the original message carried, such as the sender, the target, the method selector, the arguments (real parameters given to the message sending expression), the reply destination (where to send reply to the message), etc. In this sense, a message continuation can be regarded as a *representation* of the original message. Message continuation framework allows the programmer to reason about the information carried by a message, to store the representation of a message in data structures, and to reproduce a message using the information contained in a message continuation.

Another task of message conversion is to create a new message that carries the message continuation as its argument and delivers to some specified target. We call this new message the **message continuation carrier** of the message continuation. The method selector of a message continuation carrier is always Cont. Thus, when a message continuation carrier is accepted by the specified target, it invokes a method named Cont[2]. Because a message continuation carrier contains a message continuation as its real argument, the message continuation is passed to the Cont method on its invocation.

Like other objects, the information kept by a message continuation is encapsulated and the programmer does not have direct access to the implementation of a message continuation. The programmer can manipulate a message continuation indirectly through three primitive operations which are defined as methods of MsgCont class, namely fire, copy, and reply:

Reactivation — "$k \leftarrow$ fire()"

Reactivation operation produces a message which carries the same information as is represented by the message continuation, k.

Copy — "$k \leftarrow$ copy(*target*)"

Copy operation creates a copy of the message continuation, k. The argument to this operation (*target*) specifies the target for the copy. Multicasting (Subsection 4.2) is realized using this feature. If **Nil** is given for the argument, the target of the copy is the same as the original message continuation (k).

Forced Reply — "$k \leftarrow$ reply(*result*)"

Forced reply operation on a message continuation sends reply to the reply destination contained in the message continuation, k. It is possible to send reply in advance of reactivation of the

[2] HARMONY/2 has more sophisticated naming mechanism and pattern matching features but we will only give an outline in this article.

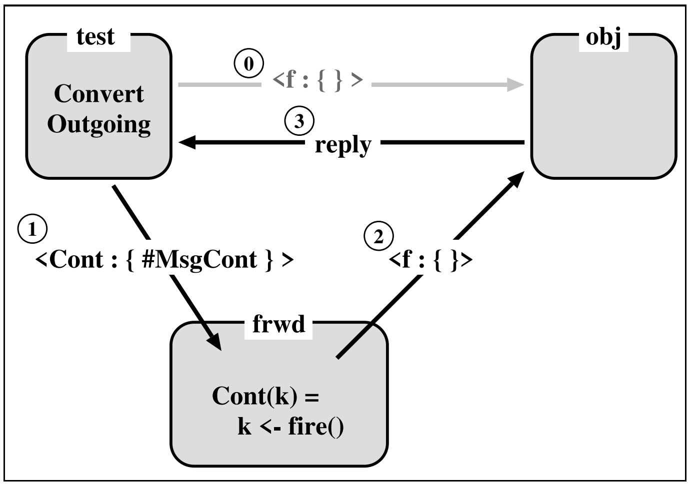

Figure 1: Example 1

message continuation. In this case, forced reply releases the blocked sender before the message continuation invokes the method with the future reactivation, thus allowing the original sender and the receiver to execute concurrently. This asynchrony is used in the implementation of asynchronous communication in Subsection 4.1.

Programming with message continuations consists of two parts: specification of message conversion and definition of Cont method.

Message conversion is specified in per-class basis using two special forms **Convert Incoming** form and/or **Convert Outgoing** form. "**Convert Incoming** $\langle$*pattern*$\rangle$ $\Rightarrow$ *target*", specifies message conversion to take place for every incoming message that matches the pattern ($\langle$*pattern*$\rangle$ $\Rightarrow$ *target*). The pattern can be either a method name, which matches a message having the same name for the method selector, or unspecified ('*'), which matches any message. If a message delivered to the instance of the class matches the pattern, the message is converted and the message continuation is delivered to the specified target (*target*), by a message continuation carrier. Another special form, "**Convert Outgoing** $\langle$*pattern*$\rangle$ $\Rightarrow$ *target*" has the same effect except that it applies to the messages that are sent out of the instance by message passing.

A Cont method is defined just like an ordinary method using **mth** method definition form. The programmer can manipulate the message continuation by the three primitive operations. Also, like an ordinary object, the message continuation can be saved in an instance variable or be sent to another object.

3.2 Examples

This subsection shows two very simple examples of the use of message continuations. Though, they are not practically very interesting, they capture essence of programming with message continuations. With slight modification to these examples, we can implement asynchronous communication, multicasting, object aggregate, message monitoring systems, and so on.

Example 1

This example explains *message conversion* and *reactivation operation* of a message continuation (Figure 1). The class definition below declares a class named Test. It declares two instance variables, *frwd* and *obj*, and one method, m. **Convert Outgoing** declaration specifies every message which has f as its method selector to be converted into a message continuation, and carried to the object referenced by *frwd*.

Class Test
    var *frwd* : Forward;
    var *obj* : SomeClass;
    Convert Outgoing f $\Rightarrow$ *frwd*;
    mth m() = *obj* $\leftarrow$ f()
EndClass

Suppose an instance of Test class, *test*, executes method m and sends a message to *obj*. If it were not for **Convert Outgoing** declaration, invocation of m method simply results in message passing to the object which is referenced by *obj* (⓪ in Figure 1). In case of this example, message conversion takes place; the message is converted into a message continuation, and delivered to *frwd* by a message continuation carrier (①). The class definition of *frwd* is the following:

Class Forward
    mth Cont(k : MsgCont) = $k \leftarrow$ fire();
EndClass

The message continuation carrier invokes Cont method, passing the message continuation as the real argument (k). (In the subsequent programming examples, k and k' denote message continuations). Cont method simply reactivates the message continuation by fire method and produces an identical message with the message produced by the message passing in the body of method m (②). The message is delivered to the original target (*obj*) and invokes method f as if the message had originated from *test* and were directly passed to *obj* (②). The reply from *obj* is sent back to the original sender (③) instead of the reactivator (i.e., *frwd*). In this example, Cont method does nothing but to reactivate the message continuation. Its behavior is analogous to invocation of Scheme's continuation immediately after calling *call/cc*.

Example 2

This example explains *copy operation* and *forced reply operation*. The implementation scheme is illustrated in Figure 2. The definition of Test class remains the same. Forward class is redefined as below. This version of Cont method makes a copy of the message continuation (k'), reactivates the copy (②), and then replies back to the original sender by forced reply operation (reply) on k (③). As will be explained in detail in the next subsection, behavior of reactivation operation and treatment of *obj*'s reply differ from those given in Example 1. When a copy of a message continuation is reactivated, as in this example, the reply is not sent to the original sender (*test*) but to the object that reactivated the message continuation (*frwd*).

Class Forward
    mth Cont(k : MsgCont) =
        let val k' = $k \leftarrow$ copy();
            val *res* = $k' \leftarrow$ fire()
        in $k \leftarrow$ reply(*res*)
        end
EndClass

3.3 Soundness

The message continuation framework provides concurrent languages with powerful extensibility in describing communication protocols and synchronization schemes. However it is not desirable to make the framework so powerful that it could change the semantics of built-in primitives and crash the system. To make our framework safe, we would like to minimize the effect of extensibility features upon semantics of built-in communication primitives. To achieve this goal, we categorized message continuations into two different kinds, namely **asynchronous message continuations** and **synchronous message continuations**, and gave them different semantics. The most interesting result obtained from this categorization is that the framework does not change the following three essential properties of synchronous

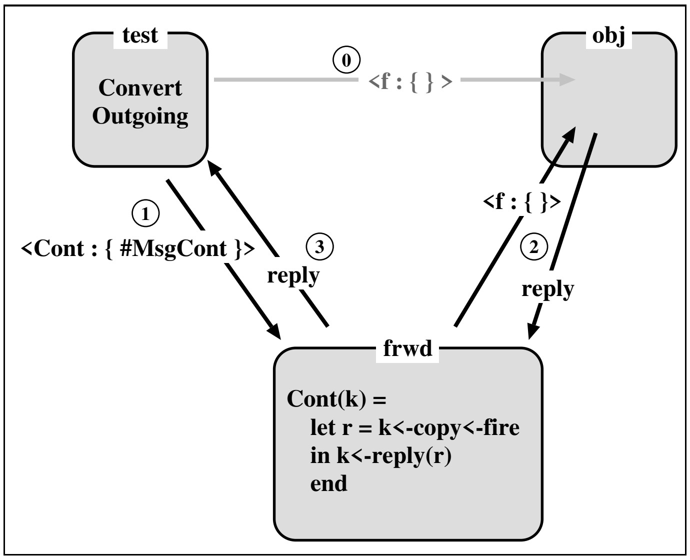

Figure 2: Example 2

communication:

Ⓐ *The sender of a message receives at most one reply.*

In normal synchronous message passing, method execution sends reply at most once. If multiple replies are sent, the sender receives the first one and the rest are never received because the sender continues execution of the method code.

Ⓑ *Every message has its reply destination.*

In normal synchronous message passing, the sender waits until the receiver completes method execution and sends reply back. If reply destination is not specified for a message, the receiver (or the method execution) cannot reply.

Ⓒ *Every message has exactly one method execution waiting for the reply of the message.*

In normal synchronous message passing, the sender waits until it receives reply. This property avoid a situation where the method execution specified by the reply destination does not wait for the reply and thus the receiver of the message cannot send reply.

In our message continuation framework, each message continuation is either an asynchronous message continuation or a synchronous message continuation. An *asynchronous message continuation* is the one which is created by message conversion. Note that message conversion stores the original sender of the message in the message continuation among other information (e.g., target, method selector, etc.) and that the original sender of the message waits from the reply coming from either reactivation or forced reply of the message continuation. A copy of a message continuation is always a *synchronous message continuation*. No method execution waits for reply from reactivation of the message continuation and thus the reply destination is not specified[3]. In contrast to an asynchronous message continuation, no method execution waits for reply from reactivation of a synchronous message continuation because it was not created by message conversion, and thus, its reply destination is not specified. In the examples of the previous subsection, k is an asynchronous message continuation because it is made from message conversion whereas k' is a synchronous one because it is a copy of a message continuation. Reactivation and forced reply operations are given different semantics depending on the category of the message continuations. The two categories took their name after their asynchronous and synchronous behavior on reactivation operation, respectively.

A message continuation obtained from message conversion, namely an asynchronous message continuation, has the original sender waiting for the reply. There are two choices in the treatment of the reply

[3]Note: This is not against property Ⓑ. Ⓑ applies to messages but not to message continuations

destination of the reactivated message: treating either the reactivator of the message continuation (e.g., *frwd* in Figure 1) or the original sender (e.g., *test*) to be the reply destination. With the former design the reactivator waits for the reply and the reactivator is responsible for sending the reply back to the original sender by forced reply operation. With the latter design the reactivator does not wait for the reply because the reply is sent directly back to the original sender. Reactivation of an asynchronous message *immediately* results in **Nil** and the method that performed reactivation continues the rest of the method execution. Thus method execution of the reactivator and the processing of the reactivated message execute concurrently. We chose the latter design to take advantage of increased concurrency. This asynchronous behavior is the origin of the name of the '*asynchronous* message continuation'.

To guarantee the *at most one reply* property (property Ⓐ), the semantics of the message continuation should change after the first reactivation. Otherwise, more than one reply may be sent by repeated forced reply and/or reactivation the same message continuation. In the framework, reactivation operation (fire method) has side effect. It changes an asynchronous message continuation into synchronous one which does not send reply to the original sender upon subsequent reactivation or forced reply. From an implementation point of view, the side effect simply nullifies the reply destination kept in the asynchronous message continuation. It ensures property Ⓐ for repeated reactivation of the same asynchronous message continuation. From the same reason, reply method also leaves similar side effect.

The reply destination of a synchronous message continuation is nullified. If reactivation of a synchronous message continuation simply produces a message using the information kept in the message continuation, the reactivated message will have no reply destination. This is against property Ⓑ. For this reason, in the framework, reactivation assigns the reactivator (*frwd* in Figure 2) to be the reply destination of the reactivated message. Consequently, the reply of processing the reactivated message goes to the reactivator. Reactivation of a synchronous message continuation blocks and waits for the reply. This synchronous behavior of reactivation of a synchronous message continuation and treatment of the reply for it is depicted in Figure 2. Forced reply operation on a synchronous message continuation is meaningless because no reply destination is nullified. It does nothing and simply results in **Nil**.

By careful design of the semantics of primitive operations on message continuations the framework guarantees the three properties of synchronous message communication (properties Ⓐ, Ⓑ, and Ⓒ). The following is a rough sketch of the proof of soundness:

Ⓐ In normal message passing, a sender receives at most one reply from the target of the message.[4] A sender never receives reply from reactivation or forced reply of a synchronous message continuation because the reply destination of a synchronous message continuation is always nullified. A synchronous message continuation may send at most one reply to the sender because it changes to an asynchronous message continuation after the first reactivation or forced reply.

Ⓑ In case of normal message produced from message passing, the reply destination is the sender. Reply destination of a reactivated message is either the original sender, in case of an asynchronous one, or the reactivator, in case of a synchronous one. So in both cases, a reactivated message has reply destination.

Ⓒ For normal message and a reactivated message produced from an asynchronous message continuation, the (original) sender waits for the reply. For a reactivated message produced from a synchronous message continuation, the reactivator waits for the reply.

In the previous design of message continuation framework, we did not consider soundness. One of the flaws with the earlier design were that an object may receive multiple replies for single message passing because of repeated forced replies and/or reactivation of the same message continuation. In the correct framework, this never happens because an asynchronous message continuation becomes synchronous after the first reactivation or forced reply operation.

The reader might think that the three properties are too restrictive, and would spoil extensibility. However, the properties only restrict the use of messages, and they don't constrain the use of message continuations. For instance, multicasting can be implemented by multiple reactivation of a message continuation; in this case what the message continuation framework guarantees is that "the original

[4]It may be the case that the sender never receives the reply due to synchronization or an infinite loop in the method definition at the target side, though.

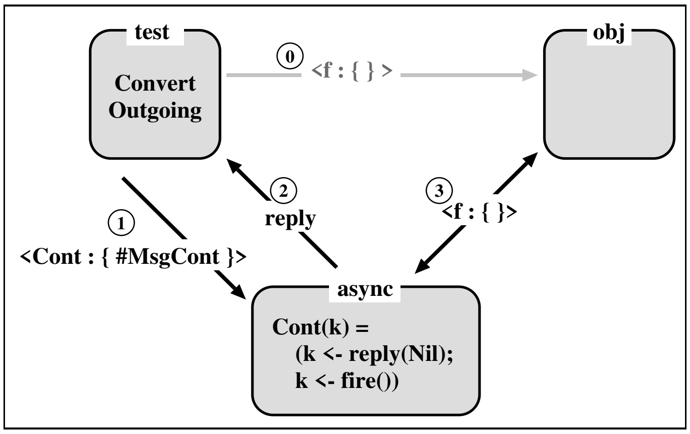

Figure 3: Asynchronous Message Passing

sender of the multicasted message receives at most one reply". As we will see in Subsection 4.2, the sender receives the reply for the first reactivation operation and replies for the subsequent reactivation operations are sent to the object that implements the multicasting protocol.

4 Practical Examples

This section presents how message continuations can be used to implement various communication protocols.

4.1 Asynchronous Message Passing

In RPC-like synchronous communication, the sender blocks and waits for a reply while the receiver processes the message. Three major roles of synchronous communication are sending information (message sending), receiving information (receiving reply), and synchronization with message processing (blocking). In some cases, not all these functionalities are necessary. Asynchronous message sending is a form of communication where only sending information takes place. In asynchronous communication, the receiver processes the message while the sender executes subsequent method code. There is no synchronization nor reply. Many object-based concurrent languages, including HARMONY/2, have built-in support of asynchronous communication [19, 20, 10, 6, 16].

In this subsection, we implement asynchronous communication using a synchronous message passing primitives and message continuations. The implementation scheme is depicted in Figure 3. We add **Convert outgoing** declaration in the class definition of the sender. Message conversion takes place and a message whose method selector is f is sent out of *test* object and converted into a message continuation. A message continuation carrier is also created and delivers the message continuation to an object named *async*, which is an instance of Asynchronous class (①).

```
Class Test
    var async : Asynchronous
    var obj : SomeClass
    Convert Outgoing f ⇒ async
    mth m() = obj ← f()
EndClass
```

Asynchronous class implements asynchronous communication. Cont method first performs forced reply to release the blocking sender, and let it execute subsequent method code (②). Then it reactivates the message continuation (③). As k becomes a synchronous message continuation by the forced reply operation, the reply for the reactivated message is sent to *async* (③).

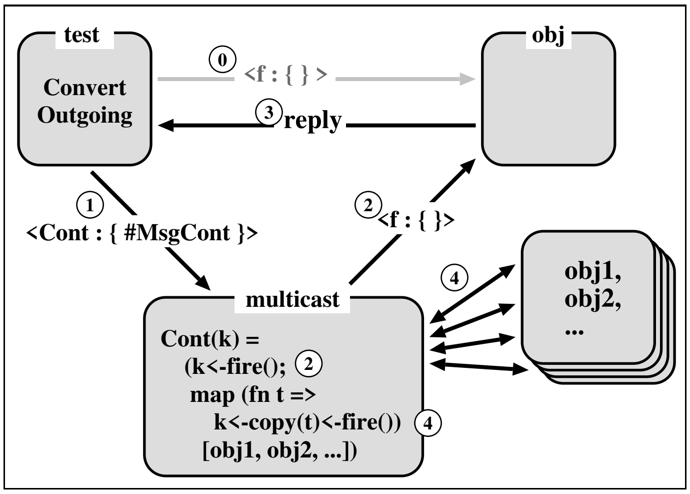

Figure 4: Serial Multicasting

Class Asynchronous
  mth Cont (k : MsgCont) =
    ($k \leftarrow$ reply (**Nil**)
     $k \leftarrow$ fire ( ))
EndClass

It is well-known that asynchronous communication can be implemented by buffered synchronous communication. However, to achieve asynchrony, we need to modify the method definitions of all the classes that participate in asynchronous communication by adding some code which implements asynchronous protocol on both the sender side and the receiver side. In contrast, by using message continuation framework, the required modification is merely an addition of conversion specification at the sender side. Because we also have **Convert Incoming** declaration, we can also add conversion specification at the receiver side, instead of adding **Convert Outgoing** declaration at the sender.

4.2 Multicasting

Multicasting is a form of communication, where copies of a message are delivered to multiple destinations. There are many variations of protocols which differ in targets specification, treatment of replies, semantics of synchronization, etc. Here we present two implementations of multicasting. First one, which we call *serial multicasting*, sends messages to multiple targets one after another. Each message is sent only after the processing of the former message completes. Second one, which we call *concurrent multicasting*, sends messages concurrently. Messages do not synchronize with each other.

Serial Multicasting

Class Multicast
  mth Cont (k : MsgCont) =
    ($k \leftarrow$ fire ( ) ;
     map (**fn** t => $k \leftarrow$ copy (t) $\leftarrow$ fire ( ))
       [obj_1 , obj_2 , . . .])
EndClass

A message continuation obtained by message conversion (① in Figure 4) is reactivated and the reactivated message is delivered to the original target, *obj* (②). Reply for the message goes to the original sender (③) because the message had originated from an asynchronous message continuation. The behavior, so far, is the same as Example 1 of Subsection 3.2. Because, up to the moment of the first reactivation, k is an asynchronous message continuation, Cont method proceeds its methods execution while *obj* processes the reactivated message. The expression, "*map* . . . [obj_1, obj_2, . . .]", implements

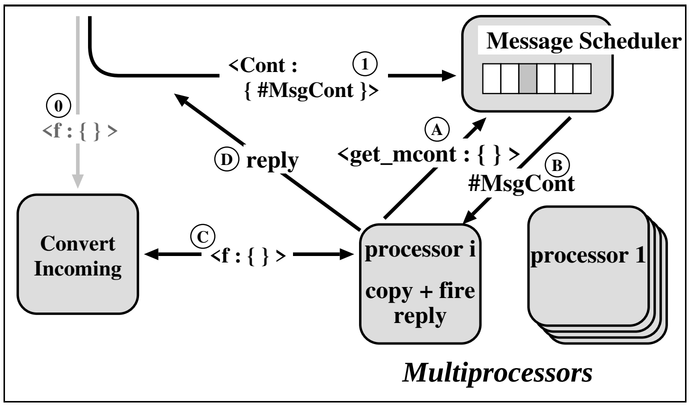

Figure 5: Emulation of Multi-processor Environment

sequential multicasting of a list of objects, "[obj_1, obj_2, ...]". The *map* function is a primitive of Standard ML which applies the function given as the first argument to the list of objects given in the second argument. Note that the destination of each copy is specified by passing the destination to copy operation ($k \leftarrow$ copy(t)). Because k becomes a synchronous message continuation after its first reactivation (②), the subsequent reactivation is performed in synchronous manner; each reactivation of a message continuation blocks and waits for reply.

Concurrent Multicasting

We can modify the code of Multicast class and make the processing of multiple copies to work concurrently. Following is the new code of Multicast class.

```
Class Multicast
    var async : Asynchronous
    Convert Outgoing Fire ⇒ self
    mth Cont(k : MsgCont) =
        (k ← fire();
         map (fn t => self ← Fire(k ← copy(t)))
             [obj1, obj2, ...])
    mth Fire(k' : MsgCont) =
        k' ← fire()
EndClass
```

In this version, reactivation of copies is not performed in the body of Cont method named Fire. Cont repeatedly creates the copies and pass them to Fire by message passing to *self*. Because of **Convert Outgoing** declaration, the message passing is performed asynchronously by an instance of Asynchronous class (see Subsection 4.1). Consequently, reactivation of copies in Fire method is performed concurrently.

4.3 Emulation of Multi-processor Environment

When we write parallel programs, we think of arbitrary number of concurrent activities and in fact we implement them as arbitrary number of logically concurrent objects. However, in reality, the amount of computing resource is bounded by the number of the processor elements and thus assignment of logically concurrent objects to physical processor elements (*processor assignment*) is one of the key issues in developing efficient concurrent systems. This subsection presents that message continuations can be used to control the number of concurrent activities and thus eases processor assignment. What we try here is to limit the number of concurrent activities by the number of processors of a parallel computer. [11] presents a solution to this problem based on reflection. The solution implements metalevel object scheduler which schedules all the threads of control running inside objects.

We present another solution using message continuations. Instead of scheduling objects, we schedule messages by a message scheduler, which manages all the messages in the system as message continuations. The computing resource is represented by processor objects. Each processor object continuously retrieves a message from the message scheduler and reactivates it.

Figure 5 illustrates the implementation scheme. The message scheduler collects all the messages in the system. For this purpose, all the incoming messages are converted and sent to the message scheduler (①). The message scheduler may be a simple FIFO buffer or may implement some sophisticated scheduling policy. We don't go into further details of the implementation of the message scheduler. The code below implements the Processor class, whose instance represents a processor element of a parallel machine. A processor object sends get_message message to the message scheduler, and retrieves the next message continuation to process (Ⓐ, Ⓑ), and then reactivates the message continuation (Ⓒ). Note that the processor object reactivates the copy instead of the one retrieved from the message scheduler. This is for the sake of synchronization with the target of the message continuation so that the processor object can detect the termination of the method execution. When the processing of the reactivated message completes, the reply is sent back (Ⓒ) and the processor object replies to the original sender by forces reply (Ⓓ) and continues all the process (Ⓐ, . . . , Ⓓ).

```
Class Processor
    fun var eval_msg (k) =
        let res = k ← copy (Nil) ← fire ()
        in k ← reply (res)
        end
    initialize (msg_sched) =
        while true
            eval_msg (msg_sched ← get_message ())
EndClass
```

5 Related Work

5.1 First Class Continuations of Sequential Languages

To the author's knowledge, Scheme [14] is the first language that incorporated first class continuation in the language specification. It provides a primitive function named *call/cc*[5]. The function *call/cc* packages up the current continuation as an *escape function* and passes it as an argument to the unary function given to *call/cc*. Invocation of the escape function gives the control to the packaged continuation. For example, in the following code, *call/cc* binds the current continuation to *k* and "(*k* **true**)" gives the control back to the continuation by calling the escape function.

```
(call/cc (lambda (k) ... (k true) ...))
```

A Standard ML compiler developed at ATT Bell laboratory, called SML/NJ, has several non-standard features which include first class continuation feature [4]. SML/NJ provides two functions, *callcc* and *throw*. The former is for obtaining first class continuations and the latter is for invocation of continuations. The SML/NJ's equivalent for the above Scheme code is:

```
callcc (fn k => ...; throw k true; ...)
```

In HARMONY/2, a message continuation is obtained by **Convert Incoming** or Convert Outgoing declaration and the message continuation is reactivated by fire operation. Message conversion and fire operation corresponds to call/cc and invocation of an escape function, respectively. Copy and reply operations of HARMONY/2 are peculiarities with concurrent languages. In Scheme and SML/NJ, we can give different names to a first class continuation by **let** binding. However those names simply share the same continuation but do not create copies. As we saw in the implementation of multicasting, copy operation creates multiple copies of message continuations which correspond to multiple threads of

[5]In the language specification of Scheme [14], the function is named *call-with-current-continuation* but most implementations of Scheme provide this abbreviation.

control. The copy operation is also used to control synchronization by producing a synchronous message continuation out of an asynchronous one. reply can change the order of method execution and reply sending , and thus increases concurrency.

5.2 Reflective Computation

From our experience with message continuations, the framework seems as expressive as a class of reflective computation called *Group-wide Reflection* [18, 11]. Examples of object migration [18] and multi-processor emulation [11], which was solved using group-wide reflectivity, can be easily described in the framework of message continuations. On the other hand, it is not simple to describe Time Warp Simulation example that appears in [11]. The example requires metalevel modeling of both objects and communication, where message continuations only feature the latter. [11] solves this using a concurrent reflective languages, ABCL/R2, which supports two dimensions of reflectivity, namely group-wide reflectivity and *individual reflectivity*.

The programs written by HARMONY/2 tends to be much shorter and more readable than the solutions described by reflective languages. One reason is that the abstraction level of message continuations is higher than ABCL/R2's metalevel message objects. Another reason is that in ABCL/R2, the programmer has to write additional protocols to protect the system from possible dangerous reflective code, while in HARMONY/2, such protection is hidden in the semantics of primitive operations on message continuations because the framework guarantees soundness.

Most reflective systems demand the programmer to follow a certain discipline, called *Meta Object Protocol*, to make the system work correctly. A problem with this approach is that when the programmer breaks the discipline, whether on purpose or inadvertently, the system can not guarantee the semantics of basic features, such as method execution, instance creation, message passing, etc. In contrast our framework is much safer in that the semantics of basic features are protected by the carefully designed primitive operations (i.e., fire, copy, and reply).

The message continuations and reflection are not conflicting concepts. They can co-exist in the same language by making message continuations as a member of the metalevel objects. Incorporation of message continuation framework in reflective computation will replace ad hoc treatment of messages in the existing reflective systems with well-founded, highly abstract primitives and improve understandability of reflective programs.

5.3 Meta Filter

Sina [2] is an object-oriented programming language which is implemented at the university of Twente. The author's previous design of first class message framework was added as Sina's Meta filter. Description power is almost compatible with message continuation framework.

Meta filter has extended features that allow the programmer to refer to or to modify information kept in the first class messages, giving more flexibility compared with message continuation framework. However, the framework of Meta filter lacks soundness. [1] discusses integration of Meta filter with other filter mechanisms that allow the programmer to implement code reuse and synchronization.

6 Concluding Remarks

We introduced the concept of continuation in object-based concurrent computation. The framework of message continuations was designed to provide concurrent languages with extensibility in modeling and programming various communication protocols. It has been used over a year to describe wide variety of communication schemes such as asynchronous communication, multicasting, sophisticated synchronization constraints, inter-object synchronization, concurrency control, resource management, and so on. In spite of its powerful extensibility, the framework is sound in that no program can destroy the semantics of built-in communication primitives. The article explained the framework using synchronous communication model. However, the framework can be also applied to asynchronous communication model. In fact, HARMONY/2 integrates both asynchronous and synchronous message passing with message continuation framework. Another advantage of the framework is that message continuations can be implemented very efficiently. Message conversion has almost no overhead and primitive operations defined for message continuations are cheap operations. A compiler of HARMONY/2 is being developed at Tokyo Institute of Technology. Features of HARMONY/2 that are not found in the article include asynchronous message passing, inheritance, and delegation. The languages is going to be used as a platform of distributed programming and inter-linguistic communication model.

References

[1] M. Aksit, K. Wakita, et al. Abstracting object interactions using composition filters. Project Report of TRESE group, the university of Twente, the Netherlands, 1992.

[2] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object-oriented langauge-database integration model: the composition-filters approach. Technical report, The university of Twente, 1992.

[3] P. America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199 – 220. MIT Press, Cambridge, Mass., 1987.

[4] A. Appel and D. MacQueen. Standard ML of New Jersey. In M. Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science, New York, August 1991. Springer-Verlag.

[5] N. S. Barghouti and G. E. Kaiser. Concurrency contorl in advanced database applications. *ACM Computing Surveys*, 23(3):269–318, September 1991.

[6] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Object-Oriented Programming Systems, Languages and Applications*, volume 21(11), pages 78 – 86. SIGPLAN Notices (ACM), November 1986.

[7] Peter de Jong, editor. *Conference on Organizational Computing Systems '91*. ACM press, 1991. SIGOIS Bulletin vol. 12, number 2 (3).

[8] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of International Conference on the Management of Data, SIGMOD RECORD*, pages 399–407, Portland, Oregon, June 1989.

[9] C. T. Haynes and D. P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, April 1987.

[10] Y. Ishikawa and M. Tokoro. Orient84/K: An object-oriented concurrent programming language for knowledge representation. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 159 – 198. MIT Press, Cambridge, Mass., 1987.

[11] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In P. America, editor, *Proceedings of European Conference on Object-Oriented Programming '91*, volume 512 of *Lecture Notes in Computer Science*, Geneva, Switzerland, July 1991. Springer-Verlag.

[12] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

[13] O. M. Nierstrasz. Active objects in Hybrid. In *Object-Oriented Programming Systems, Languages and Applications*, volume 22(12), pages 243 – 253. SIGPLAN Notices (ACM), December 1987.

[14] J. Rees and W. Clinger. Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Not.*, 21(12):37–79, December 1986.

[15] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.

[16] 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.

[17] M. Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28, 1980.

[18] T. Watanabe and A. Yonezawa. An actor-based metalevel architecture for group-wide reflection. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL)*, Lecture Notes in Computer Science, Noordwijkerhout, the Netherlands, May 1990. Springer-Verlag.

[19] 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.

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

Acknowledgements The author would like to thank Mehmet Aksit and TRESE project at the University of Twente for discussion on Meta filter, to Aki Yonezawa and a number of people at the University of Tokyo for joining the discussion and giving comments on the earlier design of message continuation framework and its soundness. Hidehiko Masuhara at the university of Tokyo kindly explained the specification and implementation of ABCL/R2. Shigeru Chiba, Shigekazu Inohara, Hiroyuki Matsuda, Hiroto Morizane, Satoshi Matsuoka, Nobuyuki Tomizawa read the earlier drafts and gave me fruitful comments. Finally, I would like to address my thanks to colleagues at Tokyo Institute of Technology for their kindness and support.