ソフトウェア基礎論　22－1
（１９８７. １０. １）

並列オブジェクト系の変換とその応用

柴山悦哉

東京工業大学理学部情報科学科

同期／非同期のメッセージ通信を行いながら並列かつ協調的に問題解決を行うオブジェクト系に対して、いくつかの変換規則を与える。これら変換規則の正当性は、イベント・ダイアグラムを用いて定義される。ここで、イベント・ダイアグラムとは、オブジェクト系で発生したイベントとこれらイベント間の因果関係の対である。変換規則の正当性も、このイベント・ダイアグラムに基づいて証明される。

本稿では、並列オブジェクト系の変換規則の応用例として、二分探索木のある実現方式から別の実現方法を導く。

我々の変換技法は、主として並列オブジェクトの融合と分離を行うものである。

Program Transformation and Its Applications in an Object-Based Parallel Computing Model

Etsuya SHIBAYAMA
Department of Information Science,
Tokyo Institute of Technology,
Oh-Okayama, Meguro-ku, Tokyo, Japan, 152

Several transformation rules are presented which are applicable to systems of concurrent objects, i.e., those which are constituted of computational agents with capability of concurrent execution and message passing. The correctness of these transformation rules are defined and also proven based on event diagrams, each of which is a pair of events and the causality relation on them. We also present an application example of the rules: inducing an implementation scheme of a binary search tree from another implementation scheme. Our approach is mainly based on fusing and splitting concurrent objects.

1 Introduction

Current computer technology offers network connected computing systems consisting of tens to hundreds of processing units in a comparatively easy and reasonably cheap way. These forms of systems are widely used at laboratories, factories, offices, and so on. The rapid progress of them seems to promise well for the future. However, widely and advanced use of distributed systems requires more and more distributed software, whose design and development are not easy under the today's software technology.

An object oriented concurrent programming paradigm is expected to offer a methodology which supports design and development of correct, efficient, and large distributed software. A computation model based on this paradigm consists of computational agents called *objects*, which have a capability of concurrent execution and transmission/reception of messages. Nothing except the objects are active and nothing except the messages can be used as communication and/or synchronization facilities. This kind of models are well suited for modelling distributed systems.

From a theoretical point of view, transformation (or an algebraic treatment) of concurrent object systems is more difficult than that of functional and logic programming languages. The behavior of an object depends not only on the recently received message but also on its internal state which may be varied during execution. This is not the situation of functional and logic programming languages.

If transformation techniques are available for concurrent object systems, they will be great help for optimization, verification, and development of distributed programs. This paper describes a first step towards a transformation system for a concurrent object oriented language.

2 The Computation Model

In the object oriented computing model assumed in this paper is similar to the one which is proposed in [Yonezawa et al. 86] and [Yonezawa and Shibayama 87]. In the model, each object has its own computing power and can perform its work concurrently with the other objects. The computation model includes neither shared memory nor global clock. Objects constitute a sparsely connected system.

An object has its own internal world, which consists of a local persistent memory and procedures inquiring/updating the local memory. They are called *state* and *script*, respectively. We assume that an object has its own local clock. The internal world of an object cannot be accessed directly from the other objects.

Objects interact with one another via *message passing*. In response to a message, an object executes one of the procedures in its script. Execution of a procedure by an object consists of a sequence of actions:

- inquiring and updating the *state* of the object,
- creating new objects,
- sending messages and receiving replys, and
- returning a value as a reply to a received message.

Messages arriving at an object will be processed one at a time in a sequential manner. More precisely, while an object executes a procedure, messages cannot arrive at the object.

There are two types of message passing, *asynchronous* and *synchronous*. Just after sending an *asynchronous* message, an object can continue its computation. Concurrently, if the receiver is not busy at that time, it accepts and processes the message. The sender object of an asynchronous message expect no reply. In contrast, with *synchronous* message passing, the sender object cannot resume its computation until the reply to this message arrives.

Objects have two kinds of *ports*, *message ports* and *reply ports*. Each object has exactly one message port. Each message sent to the object will arrive at this port. An object may create a reply port dynamically. The reply to a synchronous message will arrive at a reply port and the arrival of the reply triggers the resumption of the object.

Each synchronous message implicitly contains the information about the place to which its reply should be returned back. We call this information as *reply destination*. On synchronous message transmission, an object first creates a *reply port* and then sends a message with specifying the port as the reply destination. The receiver object of a synchronous message can forward the reply destination to another object, which is, in this case, responsible to returning back a reply. In our computation model, a reply is just a asynchronous message to a reply destination.

Messages satisfy the *transmission ordering law*: Suppose that two messages M and M' are transmitted by the same sender object in this order according to the local clock of this object. If M and M' have the same arrival port, they arrive in the same order according to the local clock of the receiver. In general, if M and M' are sent from the different objects and received by the same one, their arrival order cannot be determined.

3 The Description Language

We will describe the state and script of an objet according to the syntax of our object-based concurrent language ABCL/1 [Yonezawa et al. 86], [Shibayama and Yonezawa 86]. In the language, objects are defined in the following form:

```
[object <object-name>
  (state [<state-variable> := <initial-value>] ...)
  (script
    (=> <message-pattern> @ <reply-destination>
        <behavior-description>  ...)
              ...)]
```

The `<state-variables>`s of an object are the variables which represent the internal *state* of the object. An object defined in the above form accepts a message matching some `<message-pattern>`. At that time, the reply destination of the message (if it exists) is bound to the corresponding `<reply-destination>` which is a temporary variable. After accepting a message, the object will take a sequence of actions described in the `<behavior-description>`s folloinwg the `<reply-destination>`. The state variable declaration part and `<reply-destination>` are optional.

In the `<behavior-description>`s, message passing and returning a reply are described in the following forms.

```
[<object> <== <message>]       synchronous message passing
[<object> <= <message>]        asynchronous message passing
[<object> <= <message> @ <reply-destination>]
          asynchronous message passing with reply destination
!<reply>  returning a reply
```

Sequential computation within an object is described using lisp-like forms.

4 Correctness of Transformation Rules

In order to discuss transformation of object systems, we have to define the correctness of transformation rules. For this purpose, we firstly define *events* and *event diagrams*.

Definition 4.1 *An event is one of the following.*

- *transmission of a message*
- *acceptance of a message*

For a set of events E, we define obj_E and $time_E$ as follows:

- *If $e \in E$ is the transmission of a message, $obj_E(e)$ is the object which transmits the message.*
- *If $e \in E$ is the acceptance of a message, $obj_E(e)$ is the object which accepts the message.*
- *$time_E(e)$ is the $obj_E(e)$'s local time when e occurs.*

Definition 4.2 *A computation is a tuple $\langle E, obj_E, time_E\rangle$ where E is a set of events each of whose acceptance events has the corresponding transmission event in E.*

Definition 4.3 *Let $C = \langle E, obj_E, time_E\rangle$ be a computation. The causality relation $\overset{C}{\Rightarrow}$ of C is the transitive closure of the union of the following relations $\overset{obj}{\Rightarrow}$ and $\overset{mess}{\Rightarrow}$:*

- *$e \overset{obj}{\Rightarrow} e'$ if and only if $obj_E(e) = obj_E(e')$ and $time_E(e) < time_E(e')$*
- *$e \overset{mess}{\Rightarrow} e'$ if and only if e is the transmission event of a message and e' is the acceptance event of the same message*

Definition 4.4 *Let $C = \langle E, obj_E, time_E\rangle$ be a computation. An event diagram corresponding to C is a pair $\langle E, \overset{C}{\Rightarrow}\rangle$.*

If C is a realizable computation, the event diagram corresponding to C must be a directed acyclic graph. We consider a computation of an object system is represented as the corresponding event diagram. The semantics of an object system is defined as the set of the possible event diagrams.

Figure 1 is a pictorial representation of an event diagram. In this figure, each vertical line represents temporal axis of an object A or B. and each sloping arrow represents message passing. The small circles are the events on A and B. The object A accepts a message and transmits two messages to B, whereas the object B accepts two messages from A and transmits two messages. The event diagram represented by this figure is the transitive closure of this graph.

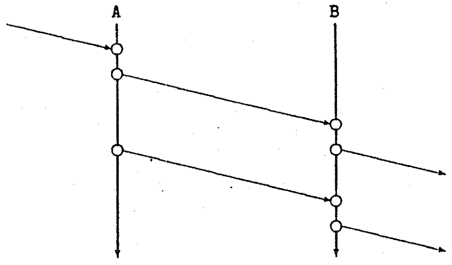

Figure 1: An example of a pictorial representation of an event diagram

Definition 4.5 *An event diagram $\langle E_1, \overset{C_1}{\Rightarrow}\rangle$ subsumes an event diagram $\langle E_2, \overset{C_2}{\Rightarrow}\rangle$ if there exists a one-to-one mapping $f \in E_1 \rightarrow E_2$ which satisfies the following conditions.*

1. *if $e \in E_1$ is the transmission (or acceptance) event of a message, $f(e)$ is also the transmission (or acceptance, respectively) event of the same message.*
2. *For $e \in E_1$, $obj_{E_1}(e) = obj_{E_2}(f(e))$.*
3. *$e \overset{C_1}{\Rightarrow} e'$ if $f(e) \overset{C_2}{\Rightarrow} f(e')$.*

Two event diagrams are isomorphic if and only if they subsume each other.

Intuitively speaking, if an event diagram $E_1, \overset{C_1}{\Rightarrow}$ subsumes another event diagram $E_2, \overset{C_2}{\Rightarrow}$, the degree of nondeterminism of C_1 is higher than or equal to that of C_2.

Next, we will define a restriction on an event diagram. This is a similar concept of the restriction of CCS [Milner 80], [Milner 85]. However, because of the asynchronous nature of our model, the definition is different.

Definition 4.6 *Let $C = \langle E, obj_E, time_E\rangle$ be a computation and S be a subset of $obj_E(E)$.*

$$\begin{aligned} \langle E, \overset{C}{\Rightarrow}\rangle|_S &\overset{def}{\equiv} \langle E', \Rightarrow'\rangle \text{ where} \\ E' &= \{e \in E | obj_E(e) \notin S\}, \\ e_1 \Rightarrow' e_2 & \text{ if and only if } e_1 \overset{C}{\Rightarrow} e_2 \text{ and } e_1, e_2 \in E' \end{aligned}$$

$\langle E, \Rightarrow\rangle|_S$ is the restriction of $\langle E, \Rightarrow\rangle$ on $obj(E)_E - S$. The correctness of a transformation is defined in terms of event diagrams.

Definition 4.7 *Let S be a set of objects. Suppose that a subset S_1 of S is transformed into S_2.*

- *This transformation is correct as implementation if, for each event diagram $\langle E', \overset{C'}{\Rightarrow}\rangle$ corresponding to a computation posterior to the transformation, there exists an event diagram $\langle E, \overset{C}{\Rightarrow}\rangle$ corresponding to a computation prior to the transformation such that $\langle E, \overset{C}{\Rightarrow}\rangle|_{S_1}$ subsumes $\langle E', \overset{C'}{\Rightarrow}\rangle|_{S_2}$.*
- *This transformation is correct if it is correct as implementation and the transformation in the reverse direction is also correct as implementation.*

By an application of a transformation rule which is correct as implementation, the degree of nondeterminism of an object system may be reduced. Therefore, in verification, we must carefully examine whether transformation rules are correct or only correct as implementation. In implementation, however, transformation rules which are not correct but correct as implementation are sometimes useful. Note that, every correct transformation rule is reversible. That is, the transformation in the opposite direction is also correct.

5 Fusing/Splitting Objects

In this section, we show some transformation rules and prove their correctness. These transformation rules fuse objects into one or split an object into many.

Rule 5.1 is an example of a correct transformation rule, which merges two objects into one.

Rule 5.1 *Precondition: Two objects* A *and* B *satisfy the following conditions.*

1. *All the messages that arrive at the message port of* B *are transmitted by* A.
2. *All the messages transmitted by* A *arrive at the message port of* B.
3. *For each synchronous message transmission from* A*,* B *always returns back a reply.*

Transformation: Fusing A *and* B *into the object, say* A+B*, such that:*

1. *The state of* A+B *is the disjoint union of those of* A *and* B*. In the case that* B *is bound to some state variable of* A*, which is used as the destination of message passing to* B*, the state variable will be removed from the state of* A+B.
2. *The message patterns of* A+B *is same as those of* A.
3. *The behavior of* A+B *is obtained from* A*'s behavior by substituting each message transmission by* B*'s corresponding behavior.*

An application example of Rule 5.1 is illustrated in Figure 2. Two objects generator and filter are fused into a single object generator+filter. In the figures, output is an external object.

Figure 3 represents an event diagram prior to the transformation and Figure 4 represents the corresponding event diagram posterior to the transformation. In these figures, each white circle is an internal events within generator and filter, whereas each black circle is an external one. The causality relation among the events represented as the black circles are preserved.

Rule 5.1 is easily proven to be correct.

Theorem 5.2 *Rule 5.1 is correct.*

Proof: *Let S_1 be $\{$A, B$\}$ and $\langle E, \Rightarrow\rangle$ be an arbitrary event diagram prior to the transformation. We can show that there exists an event diagram posterior to the transformation such that its restriction on $\{$A + B$\}$ is isomorphic to $\langle E, \Rightarrow\rangle|_{S_1}$. Let $m_1, ..., m_n$ be the messages accepted by* A *in this order in $\langle E, \Rightarrow\rangle$. In response to m_i $(1 \le i \le n)$,* A *sends messages,*

```
[object generator
  (state [n := 1])
  (script
    (=> :start
        (loop
          [filter <= n]
          [n := (+ n 1)]))))]

[object filter
  (script
    (=> number
        (if (evenp number)
            [output <= number]))))]
```

$$\Downarrow$$

```
[object generator+filter
  (state [n := 1])
  (script
    (=> :start
        (loop
          (if (evenp n) [output <= n])
          [n := (+ n 1)]))))]
```

Figure 2: Fusion of generator and filter

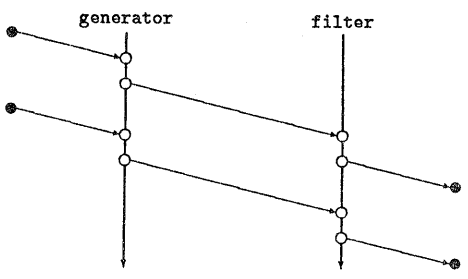

Figure 3: Before an application of the transformation rule

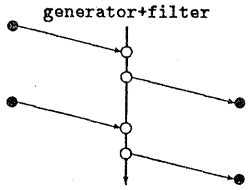

Figure 4: After an application of the transformation rule

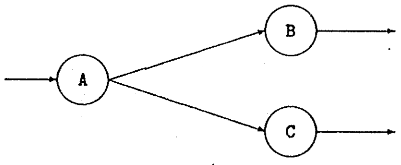

Figure 5: Object subsystem with three objects

say $m_{i,1}, ..., m_{i,j_i}$ in the transmission order, to B. B *also transmits messages in response to $m_{i,j}$ $(1 \leq i \leq n, 1 \leq j \leq j_i)$, say $m_{i,j,1}, ..., m_{i,j,k_{i,j}}$ in this order. We assume that, on transmission of $m_{i,j}$,* A*'s state is $a_{i,j}$ and that, on transmission of $m_{i,j,k}$,* B*'s state is $b_{i,j,k}$. Then, if* A+B *receives the messages $m_1, ..., m_n$ in this order, we can easily proven by induction that:*

1. *the $(k + \sum_{j'=1}^{j} k_{i,j'} + \sum_{i'=1}^{i-1} \sum_{j'=1}^{i'_j} k_{i',j'})$-th message* A+B *transmits is $m_{i,j,k}$ and*
2. *at that time the state of* A+B *is the disjoin union of $a_{i,j}$ and $b_{i,j,k}$.*

Notice that A *is deadlock free from the precondition of the transformation rule and that, if* B *falls into deadlock,* A+B *also falls into deadlock just after the same message passing.*

It is proven in the same way that each event diagram $\langle E, \Rightarrow \rangle$ posterior to the transformation has an corresponding diagram $\langle E', \Rightarrow' \rangle$ prior to the transformation such that $\langle E, \Rightarrow \rangle|_{\{A+B\}}$ is isomorphic to $\langle E', \Rightarrow' \rangle|_{\{A,B\}}$.

Note that in this proof, n, j_i $(1 \leq i \leq n)$ and $k_{i,j}$ $(1 \leq i \leq n, 1 \leq j \leq i_j)$ can be infinite ordinal numbers. The correctness of Rule 5.1 much depends on the transmission ordering law.

In Rule 5.1, if A transmits message not only to B but also another object C (Figure 5), generally, it is not incorrect to fuse A and B into one. The reason is simple: In the case that B deadlocks during computation, a message transmitted from A to C before the fusion cannot be transmitted by A+B after the fusion.

In Figure 5, even if these three objects are deadlock free, the transformation which merges them into one is not correct in general. For instance, in the case that B and C transmit messages to the same object, say D, the acceptance order of two messages sent from B and C is arbitrary. However, by fusion of A, B, and C, the acceptance order of the same two messages becomes deterministic because of the transmission ordering law. That is, this fusion rule is at most correct as implementation.

Rule 5.3 *Precondition: Three objects* A, B, *and* C *satisfy the following conditions.*

1. *Each message that arrives at the message port of either* B *or* C *is transmitted by* A.
2. *Each message transmitted by* A *arrives at the message port of either* B *or* C.
3. A, B, *and* C *are deadlock free.*

Transformation:

```
[object generator
  (state [n := 1])
  (script
    (=> :start
        (loop
          (if [evenp <= n] then
              [n := (+ n 1)]))))]

[object evenp
  (script
    (=> number
        !(evenp number)))]

                    ⇓

[object generator
  (state [n := 1])
  (script
    (=> :start
        (loop
          (if (evenp number) then
              [n := (+ n 1)]))))]
```

Figure 6: Fusion of generator and evenp

1. *The state of* A+B+C *is the disjoin union of those of* A, B, *and* C. *In the case that* B *and/or* C *is bound to some state variable(s) of* A, *which is used as the destination of message passing, the state variable will be removed from the state of* A+B+C.
2. *The message patterns of* A+B+C *is same as those of* A.
3. *The behavior of* A+B+C *is obtained from* A*'s behavior by substituting each message transmission by the corresponding behavior of* B *or* C.

Theorem 5.4 *Rule 5.3 is correct as implementation.*
Proof: *This thorem can be proven in the same way as Theorem 5.2.*

The following is a correct transformation rule.

Rule 5.5 *Precondition: Two objects* A *and* B *satisfy the following conditions.*

1. *All the messages that arrive at the message port of* B *are transmitted by* A.
2. *All the messages transmitted by* B *arrive at reply ports of* A.
3. *For each synchronous message transmission from* A, B *always returns back a reply.*

Transformation: Fusing A *and* B *into the object, say* A+B *in the same way as Rule 5.1.*

An application example of this transformation rule is illustrated in Figure 6. Two objects generator and evenp are merged into generator+evenp.

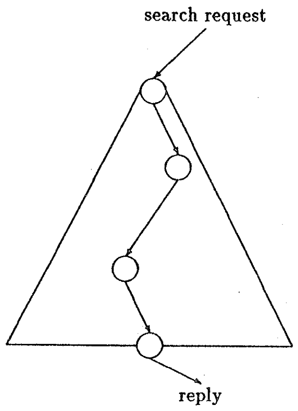

Figure 7: A binary search tree

6 An Application Example

In this section, we show the relation between two implementations of binary search trees using our transformation technique. With asynchronous message passing, a binary search tree constructed from nodes defined in the following program is natural.

```
[object CreateNode
  (script
    (=> [:new my-key my-value left right]
        ![object
          (script
            (=> [:search key] @ R
                (cond ((= key my-key) !my-value)
                      ((< key my-key)
                       [left <= [:search key] @ R])
                      ((< my-key key)
                       [right <= [:search key] @ R]))))]))]
```

Upon accepting a `[:new ...]` message, the `CreateNode` object creates a node object whose state consists of its key, stored value, the names of the left and right children nodes. By receiving a search request, a node object returns its stored value if the received key matches its stored key. Otherwise, the object forwards the message to a left or right child according to the received key value. Figure 7 illustrates a binary search tree consisting of node objects created by `CreateNode`.

Each node object can be split into 3 objects as in the following definition.

```
[object CreateNode
  (script
    (=> [:new my-key my-value left right]
        ![object
          (state
            [State :=
              [object
                (script
                  (=> :your-value?
                      ![my-key my-value left right]))]]
            [Script :=
              [object
                (script
                  (=> [:search key State] @ R
                      (case [State <== :your-value?]
                        (is [my-key my-value left right]
                            (cond ((= key my-key)
                                   !my-value)
                                  ((< key my-key)
                                   [left <=
                                     [:search key] @ R])
                                  ((< my-key key)
                                   [right <=
                                     [:search key] @ R])))))
              )]])
          (script
            (=> [:search key] @ R
                [Script <= [:search key State] @ R]))]))]
```

The new definition can be transformed into the original one by applications of Rule 5.1 and Rule 5.5. Therefore, this transformation is correct. In this stage, `Script` objects are functional, i.e., its behavior does not depends on the temporal situation. The following definitions can be obtained by merging `Script` objects into one. This transformation is easily proven to be correct as implementation.

```
[object CreateNode
  (script
    (=> [:new my-key my-value left right]
        ![object
          (state
            [State :=
              [object
                (script
                  (=> :your-value?
                      ![my-key my-value left right]))]])
          (script
            (=> [:search key] @ R
                [Script <= [:search key State] @ R]))]))]
```

```
[object Script
  (script
    (=> [:search key State] @ R
        (case [State <== :your-value?]
          (is [my-key my-value left right]
              (cond ((= key my-key) !my-value)
                    ((< key my-key)
                     [left <= [:search key] @ R])
                    ((< my-key key)
                     [right <= [:search key] @ R])))))))]
```

Next, by merging `Script` and the objects bound to its local variables `left` and `right`, the following definition is obtained.

```
[object CreateNode
  (script
    (=> [:new my-key my-value left right]
        ![object
          (state
            [State :=
              [object
            (script
              (=> :your-value?
                  ![my-key my-value left right]))]])
        (script
          (=> [:search key] @ R
              [Script <= [:search key State] @ R]))]))]

[object Script
  (script
    (=> [:search key node] @ R
        (case [State <== :your-value?]
          (is [my-key my-value left right]
              (cond ((= key my-key) !my-value)
                    ((< key my-key)
                     [Script <=
                       [:search key left.State] @ R])
                    ((< my-key key)
                     [Script <=
                       [:search key right.State] @ R]))))))]
```

In the definition above, `left.State` and `right.State` means the value of the variables `States` in `left` and `right`, respectively. This dot notation is allowed to use only at intermediate stages.

In order to show this transformation correct as implementation by Rule 5.3, we need the *type information* that the objects bound to the variables `left` and `right` are created by `CreateNode`.

Obviously, now, the script part of each node object is not necessary. Therefore, the above definition can be transformed into the following one.

```
[object CreateNode
  (script
    (=> [:new my-key my-value left right]
        ![object
          (script
            (=> :your-value?
                ![my-key my-value left right]))]))]

[object Script
  (script
    (=> [:search key node] @ R
        (case [State <== :your-value?]
          (is [my-key my-value left right]
              (cond ((= key my-key) !my-value)
                    ((< key my-key)
                     [Script <= [:search key left] @ R])
                    ((< my-key key)
                     [Script <=
                       [:search key right] @ R]))))))]
```

The last definition well models conventional implementation of binary search tree. That is, a node is nothing but passive data and a searcher visits nodes one by one from the root to the appropriate node. Figure 8 illustrates a conventional binary tree.

By the applications of transformation rules in this section, we can show the implementation scheme of a binary tree illustrated in Figure 7 potentially has more non-determinism than the one illustrated in Figure 8. Actually, owing to the transmission ordering law, when two search messages arrive

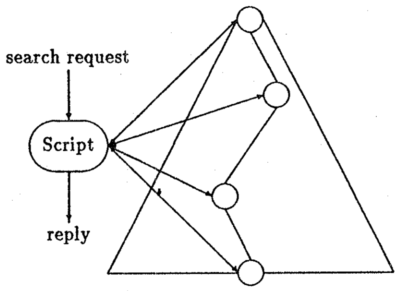

Figure 8: A binary search tree

whose answers will be found in different nodes of the same depth in Figure 8, the answers will be sent back in the arrival order of the corresponding search messages. This is not the case of the implementation scheme in Figure 8. However, without consideration of the transmission ordering of the reply messages, both implementations return back the same replys in response to the same incoming search requests.

Since this example is a read only application, the loss of non-determinism may not cause severe problems. In general, however, with a loss of non-determinism, the possible answers set may be changed. One of such phenomena is the Brock-Ackerman anomaly [Brock and Ackerman 81].

7 Conclusion

For functional/logic programming languages, we have already had the fold/unfold technique [Burstall and Darlington 77], which is often effective. However, this technique cannot directly be applicable to our object-oriented concurrent language since it is not based on neither reduction mechanism nor term re-writing system [1].

Common applications of the fold/unfold technique is fusing two or more recursive loops into one. This *loop fusion* technique is useful for the purpose of optimizations. In a concurrent object based environment, the *object fusion* technique corresponds to the loop fusion technique for function and logic based languages. In this sense, our object fusion approach offers a higher level transformation technique than the fold/unfold technique.

Our computing model and language assume parallel and distributed computing environments. Therefore, not only object fusion but also object partition are useful in order to purchase execution efficiency. The object fusion technique is useful mainly for two purposes:

- decreasing the number of message passing.
- later applications of simplification rules.

The first one is often effective on loosely coupled systems such as ethernet connected computer systems on which message passing is an expensive operation. Though the second one is important, we omit the detail of such simplification rules in the previous section since each of them is trivial and is not our concern. On the other hand, applications of the

[1]An object-oriented concurrent language can be based on reduction mechanism. Such an example is Vulcan [Kahn et al. 86].

object partition technique increase the degree of parallelism by paying communication costs.

Acknowledgements

The author would like to thank Prof. A. Yonezawa and members of the ABCL/1 project, Y. Honda and T. Takada.

References

[Agha 86] G. Agha: *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[Brock and Ackerman 81] J. D. Brock and W. B. Ackerman: Scenarios: A Model of Non-determinate Computation, *Lecture Notes in Computer Science*, Vol. 107, pp. 252–259, Springer-Verlag, 1981.

[Burstall and Darlington 77] R. M. Burstall and J. Darlington: A Transformation System for Developing Recursive Programs, *Journal of the ACM*, Vol. 24, No. 1, pp. 44–67, 1977.

[Clinger 81] W. D. Clinger: Foundations of Actor Semantics, (Ph.D. thesis), Dept. of Math., M.I.T, 1981.

[Kahn et al. 86] K. Kahn, E. D. Tribble, M. S. Miller, and D. G. Bobrow: Objects in Concurrent Logic Programming Languages, *Proceedings of Object-Oriented Programming System, Languages and Applications*, Portland, Oregon, pp. 242–257, 1986.

[Milner 80] R. Milner: *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

[Milner 85] R. Milner: Lectures on a Calculus for Communicating Systems, *Lecture Notes in Computer Science*, Vol. 197, pp. 197–220, Springer-Verlag, 1985.

[Shibayama and Yonezawa 86] E. Shibayama and A. Yonezawa: ABCL/1 User's Guide, ABCL Project, Dept. of Information Science, Tokyo Institute of Technology, 1986.

[Tamaki and Sato] H. Tamaki and T. Sato: Unfold/Fold Transformation of Logic Programs, Proceedings of 2nd International Logic Programming Conference, Uppsla, Sweden, pp. 127–138, 1984.

[Yonezawa et al. 86] A. Yonezawa, J-P. Briot, and E. Shibayama: Object-Oriented Concurrent Programming in ABCL/1, *Proceedings of Object-Oriented Programming System, Languages and Applications*, Portland, Oregon, pp. 258–268, 1986.

[Yonezawa and Shibayama 87] A. Yonezawa and E. Shibayama: Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, *Object-Oriented Concurrent Programming*, MIT Press, pp. 55–90, 1987.