A Simple Reconfigurable Object Model for a Ubiquitous Computing Environment

Kentaro Oda†, Shinobu Izumi†, Yoshihiro Yasutake†† and Takaichi Yoshida†,

†Program of Creation Informatics, Kyushu Institute of Technology, 680-4 Kawatsu, Iizuka, Japan
††Department of Intelligent Informatics, Kyushu Sangyo University, 2-3-1 Matsukadai, Higashi-ku, Fukuoka, Japan

Summary
Communication bandwidth, topology and security policy are different from place to place in a ubiquitous computing environment, components in the environment should change its behavior to fit current situation according to real world changes. Without adapting the environment, the components may fail to continue proper operations.
In this paper, we propose a reconfigurable, object model that dynamically changes the object's behavior to fit the current environment by modifying its internal structure. The proposed object consists of communicating concurrent meta-objects. Each meta-object contains functionality for adaptation, remote communication, and administration.
Generative communication, which allows meta-object communication, gives the proposed reconfigurable objects the following characteristics: flexibility; ability of allowing a variety of configurations; safety that ensures consistency; and a unification of state preservation and communication. The proposed object model was successfully implemented in a middleware framework called Juice 2.
Key words:
Reconfiguration, Adaptation, Distributed objects.

1. Introduction

A software system that modifies its behavior and structure during runtime provides high reliability and availability, and be a user-friendly system. For example, when local CPU workload becomes high, migrating an object to a remote host allows a more efficient use of hardware resources. In ubiquitous computing, as the user moves, the services offered to the user need to change seamlessly.
In the case of software evolution, updating software in runtime gives higher availability. Hardware dynamism means dynamic changes of computer resources such as CPU workload, available memory etc. Network dynamism is changes in network - such as link state, topology, bandwidth and latency etc. Software evolution and heterogeneity creates software dynamism because different version or different types of software coexists. User dynamism means the volatility of what user want to do - it sometimes depends on a context such as the physical time and place of user actually resides.

Manuscript received May 5, 2007
Manuscript revised May 20, 2007

Future computing systems should accommodate such dynamism in order to react to changes in the environment.

Our final goal is to emerge `Adaptive Software' that changes its behavior in runtime to fit current execution environment. Successful adaptive software may bring us high reliable, reboot free, resilient to unexpected changes, easy to maintain, self-optimized, and user-friendly system.
However, creating such a system is a challenge. Generally, it involves a loop that consists of the following three processes:

- Detect the underlying information about the environment;
- Decide on the appropriate action based on the information obtained;
- Perform the determined action that is needed to change the behavior of the system.

The detection of information and the decisions taken based on the information are not considered in this paper. Instead, we will focus on how a successful behavioral change is achieved. Reconfiguration in this paper means modifying the behavior by replacing the internal modules.

We define a predecessor module as the module that had been removed from the system through reconfiguration. Through reconfiguration, a successor module takes the place of a predecessor module. In general, it is conceivable that M predecessor modules have to be replaced by M successor modules. This involves integration and separation of modules. The problem arises when supporting such reconfiguration because each module uses `reference' or `pointer' for point-to-point communication, which makes the separation and integration harder. Reconfiguration becomes more difficult to achieve when the system consists of concurrent communicating modules. In this case, it is possible that a module sends a message to another module that has already been removed from the system. This leads to message loss even if they reside at the same address space. Therefore, in a reconfigurable system, it is difficult to support N to M reconfiguration, and there is no guarantee that reliable communication among the modules. The main

cause of these problems is strong assumptions among modules. Every module assumes the point-to-point direct communication and existence of the module to which a message sent at a given time.

By relaxing the inter-module assumptions, reconfiguration can be achieved by merely replacing a module. In this paper, we propose a reconfigurable object model that consists of communicating concurrent meta-objects, in which a new communication mechanism is provided to relax the assumptions. To take the modeling advantage of object orientation, we choose an object as a building block of application software and a meta-object as a replacement unit of reconfiguration.

Paper Outline

In this paper, we briefly describe the our early work, Juice 1 reconfigurable object model in section 2 and show its limitation in the next section. Section 3 presents three rules necessary for correct reconfiguration and shows the difficulty when a naive approach is applied for the reconfiguration. Section 4 gives our main idea: loosely coupled communication within a reconfigurable object. In Section 5, we give an example configuration of the proposed reconfigurable object model. Section 6 discusses about reconfiguration in the proposed model. Finally, we conclude this paper.

2. Our Early Work: A Reconfigurable Object Model - Juice 1

In our earlier proposed object model, Juice 1 model [1], is shown by Fig. 1. Each Juice 1 object consists of a context object that holds the state and methods during adaptation; a communicator object that handles the communication protocol and ordering of messages; an executor object that deals with execution control; an adaptation manager object that detects environmental information and reconfigures an entire object to an appropriate configuration given the current situation; and an encapsulation object that encapsulates the internal structure. Several items such as a new application code, communication protocols, concurrency control, as well as adaptation mechanism and policy, can, if necessary, be introduced. This makes the code for each meta-object as simple as possible. With this model, we can construct valuable objects in a distributed system. Concurrent objects, proxy objects, replica objects, and transaction-aware objects can all be introduced. Fig. 2 shows the runtime environment of this model. The testbed is written in Java since it is independent of the platform. In this testbed implementation, we introduced a pre-processor for transforming the source code to give the illusion that the application programmer does not need to give special consideration to this model.

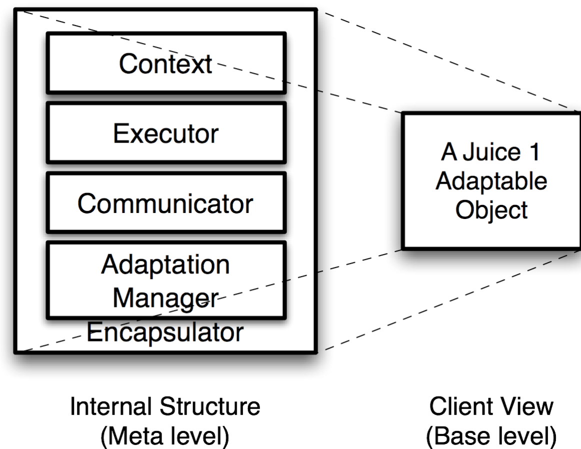

Fig. 1 Juice 1 Object Model

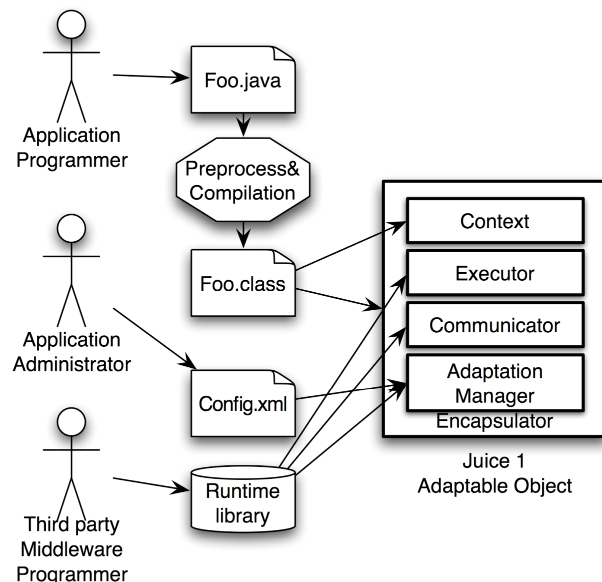

Fig. 2 Juice 1 Architecture Overview

The following is the typical internal working steps triggered by a method invocation from an ordinary Java object to a Juice 1 object.

1. On receiving a method invocation from a Java object, encapsulation object creates a message object that contains method name and parameters corresponds the invocation, and forwards it to communicator object.

2. Communicator object put the message object to a receive queue own by the communicator object. The

message object has a chance to be synchronized with external objects such as for message ordering in a multicast protocol. After this synchronization, message becomes ready to execute. Communicator object forwards this message to executor object.

3. Executor object receives the message and checks if the execution of the message ensures serializability of concurrent execution. Once serializability condition is satisfied, the executor executes a targeted method in the context object according to the information described in the message. Actual implementation of executor varies i.e. if the executor object has no multiple threads of control, no concurrency control requires.

As above steps indicate that a method invocation to Juice 1 object realized by means of internal communication among meta-objects.

By means of encapsulation object, external clients are isolated from this internal working of Juice 1 object. Juice 1 object model provides separation of concerns (SoC) through separation of a logical object into multiple meta-objects and communication among them. Each meta-object has its own state and be able to have multiple threads of control.

3. The Safe Reconfiguration Problem and Its Naive Implementation

The Safe Reconfiguration Problem

This section considers an inconsistency problem that occurs when a Juice 1 model object was reconfigured. For example, consider a reconfiguration requiring a change in the communication protocol. Not surprisingly, a naive replacement of the communicator object to the desired new protocol causes irreversible message loss since the old communicator object holds any received messages. Message loss means a loss of corresponding method execution, loss of state changes, and results in no response to the clients. This is an example of an inconsistent state. The problem of a safe reconfiguration is to transparently change an object's behavior while preventing an inconsistent state. To prevent an inconsistent state, an object must follow the following rules when replacing a meta-object:

- State consistency: The object's state must be held; it must be the same before and after reconfiguration. Every successful method invocation transfers a consistent state to another consistent state. The failure or the forced termination of method invocation can possibly cause an inconsistent state.
- Operational consistency: No operation can be removed after reconfiguration, even if its implementation is changed. However, new operations can be included during reconfiguration. The method name, parameter type, and return value type must be not changed during reconfiguration. In other words, a reconfigured object must have upwards compatibility.
- Referential consistency: Assume that object O becomes object O' after its reconfiguration. Referential consistency means that all reference to O must refer to object O' after reconfiguration. A simple solution for this problem is to keep the object's identity after reconfiguration.

Therefore, by following these rules, we can achieve a safe object reconfiguration.

A Naive Safe Reconfiguration in Juice 1

As explained in the previous sub-section, the naive reconfiguration method not only leads to a loss of information, but also causes every reference point to an obsolete object to become faulty due to lack of notification. Any operations on the obsolete objects become invalid, and thus, all references to obsolete objects are faulty. To prevent the use of a faulty reference, a two-phase commit protocol could be used. This can be described as follows (Fig. 3):

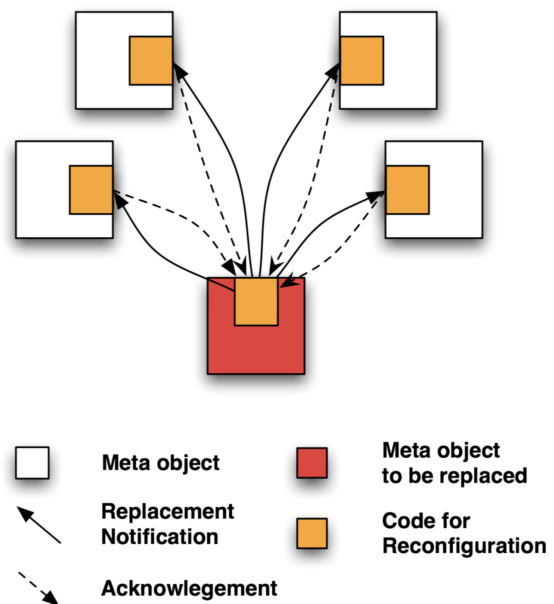

Fig. 3 Reconfiguration through negotiation among meta-objects

1. Prepare for a stop: The reconfiguration manager sends to all meta-objects a message requesting that these meta-objects prepare for the replacement of a target meta-object. In effect, this message tells them to "stop using the target meta-object";

2. Acknowledge: If the reconfiguration manager receives from all meta-objects an acknowledgment message that in effect means "I have stopped using the target meta-object. I am now ready to update." then it proceeds to the next step. Otherwise, the reconfiguration manager repeats step 1.

3. Replace and update: The reconfiguration manager creates a new meta-object and broadcasts a message announcing the new meta-object's reference. Each meta-object updates its reference according to the update message and resumes normal operations.

Basically, using this protocol, a safe reconfiguration becomes possible. However, to implement this protocol is quite difficult due to: 1) deadlock, which occurs when all the meta-objects stop, since each meta-object runs concurrently and depends on another object's work to finish; 2) low-level synchronization, which occurs in the primitive in typical object-oriented languages like Java; and 3) complexity, which crosscuts among the meta-objects. The introduction of this protocol would put a reconfigurable object into a complex, low-level, easy to deadlock, and difficult to manage state.

The main cause of these problems is that there is too tight coupling between the meta-objects. This tight coupling means that direct communication among the meta-objects is hardwired by `reference', and synchronized communication assumes the existence of two objects for communication. In reconfigurable systems, there is no guarantee that the communication target is always available. In some cases, it can happen that a communication target is split into multiple modules, or multiple modules are integrated into a single module. Direct references pin the referred object and enforce synchronous communication.

4. Introducing a Loosely-Coupled Communication For Meta-object Communication

The crux of the problem is the tight coupling among the meta-objects. If this tight coupling were to be loosened, no complex negotiation would be required, and reconfiguration could be realized by means of a simple replacement of a meta-object.

We propose a reconfigurable object model that uses loosely coupled communication for internal concurrent meta-object communication. We adopted a generative communication for loosely coupled communication because of its simplicity and flexibility. The reconfigurable object consists of a communication kernel and multiple concurrent meta-objects on top of its kernel. Generative communication is only the mechanism that allows communication among the meta-objects.

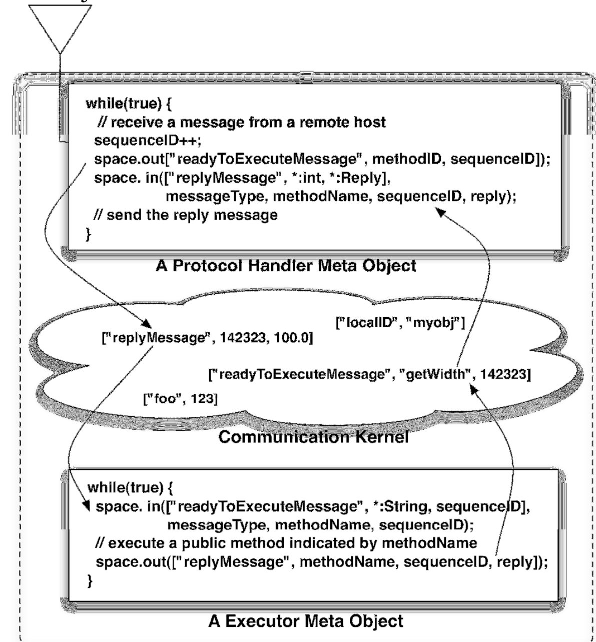

Fig. 4 The Proposed Reconfigurable Object Model

Generative Communication

Generative communication is indirect, asynchronous, content-addressing communication originally introduced in the Linda distributed programming language [2]. Generative communication requires a shared data space, called Tuple Space in Linda, where Tuple can be placed and withdrawn. In this paper, we use the term `message' instead of Tuple, and use `message space' instead of Tuple space as analogous to object orientation.

Generative communication in the proposed model allows concurrent meta-objects to exchange messages via a shared message space, into which a sender can put a message and from which a receiver can pull an expected message. Senders issue the out() operation to put a message. For example, assume that a client object calls the getWidth() method of a server object. The protocol handler's meta-object inside the server object translates this incoming call into a message ["receivedMessage", "getWidth", 123],

and submits it to a server object's internal message space. The executor object expects a message from the protocol handler by the in() operation with a message pattern called a formal message. There are two different types of messages: actual, where all the elements are concrete values; and formal, where some or all of the elements are undetermined (wild card) and bound to variables. Fig. 4 shows how the two meta-objects, protocol handler meta-object and executor meta-object, interact through a message space provided by the internal communication's kernel.

To submit the message, the protocol handler issues out() operation as follows:

```
space.out(["receivedMessage",
          "getWidth", params]);
```

The executor object expects a message from the communicator by in() operation with a message pattern.

```
space. in(["receivedMessage",
       *:String, *:Parameters],
        messageName,methodName, params);
```

The last in() operation finds a message that exact matches with a string value "receivedMessage" as the first field, an arbitrary value with “String” type as the second field, and an arbitrary value with “Parameters” type as the third and last field. Once the message pattern matches with a message in the space, in() operation returns with the actual values of a submitted message. Therefore, in this case, once received, messageType holds "receivedMessage", methodName holds "getWidth", parameters holds an object contains parameters of correspond method call. Operation out() might be used with an actual message, in which all field has a certain value, whereas operation in() might be use with a formal message, in which contains at least one field has no concrete value, i.e. "*:int", with which an arbitrary value of type “int” could match.

The condition of message match is that two messages 1) have the same length of fields, 2) have each type of field in a message equivalent and 3) are actual message and formal message or actual message and actual message. Accordingly, an actual message can be used with in() operation.

If there is no match message in the space, in() operation blocks until the submission of an expected message. Operation out() immediately returns after put the message, therefore, out() is an asynchronous operation while in() is not. On the other hand, in() and out() are atomic operation, which means, message submission and withdraw has all or nothing property, therefore, no message lost or duplication occurs.

In generative communication, any participant can withdraw the messages submitted in the space. This frees the participants from direct knowledge of when and who actually receives or sends a message. Therefore, generative communication decouples the actual time and user of the message.

5. A Reconfigurable Object Model with Generative Communication

A Sample Configuration

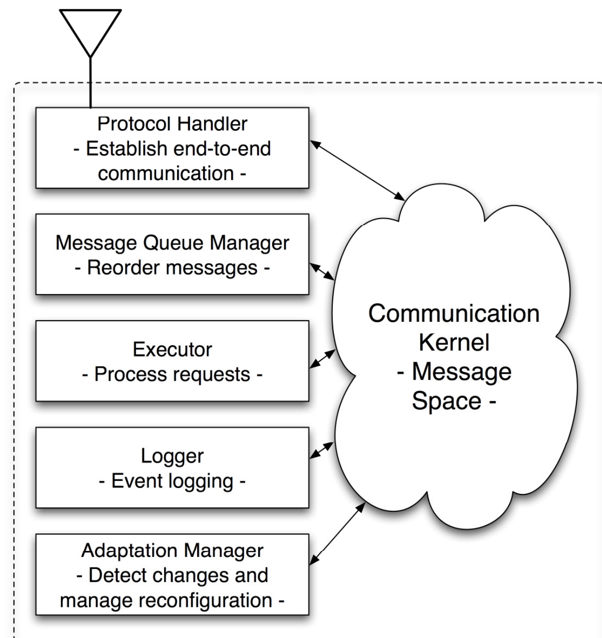

Fig. 5 A Typical Configuration of the Proposed Object Model

Fig. 5 shows a typical reconfigurable object configuration based on generative communication. The proposed reconfigurable object consists of concurrent communicating meta-objects. In the following configuration, there are five meta-objects. Each meta-object is provisioned for a specific responsibility such as end-to-end communication, re-ordering, and execution. It should be noted that this configuration is merely an example; the generality of the proposed model allows customization of this configuration.

1. End-to-end communication: A protocol handler receives all the messages coming from the outside and sends reply messages, as well as sending

request messages to the outside. In other words, this provides end-to-end communication between an external object and the reconfigurable object. The typical internal work of a protocol handler is to create a corresponding message and put it into the shared message space with the out() operation for every time it receives a message. To reply, the protocol handler waits for a reply message using the in() operation and sends it to the remote object.

2. Re-ordering: A message queue manager determines the sequence of messages ready to execute. Multicast or a two-phase commit protocol often requires synchronization between external objects. The manager allows re-ordering of received messages. To do this, the manager withdraws received messages from the space and puts the re-ordered messages back into the space. To identify re-ordered messages, a label is given by the manager before its resubmission.

3. Execution: The utilization of multiple threads might enhance performance, but it requires concurrency control. An executor takes the re-ordered messages and executes the corresponding methods concurrently or serially. After execution, the resulting message is placed into the space. An executor usually owns the methods and attributes of a reconfigurable object. Other specific executors such as transaction, persistency, debugging, and profiling, could also be provided.

4. Administration: A logger object that logs the internal activities of the reconfigurable object helps in administration and debugging. The activities include message-related events, for example, reception and execution, and reconfiguration-related events, such as detaching and attaching of meta-objects.

5. Adaptation: An adaptation manager manages reconfiguration by monitoring events of interest such as network link down, high CPU workload, or the presence of new software updates. Reconfiguration is started by sending a message requesting a targeted meta-object to be replaced. The manager installs a new meta-object. It is the responsibility of the adaptation manager to determine how and when a reconfiguration should be done in order to adapt the system to its underlying environment.

The Power of the Model: Flexibility

In the previous section, we illustrate a typical configuration. However, this model can accommodate new configurations that are specialized for each purpose. For example, we can deploy multiple executor objects to enhance performance, and hence, take advantage of parallel execution. With receiver and sender abstraction, the number of executor objects can be changed according to physical CPU resources. This model also allows multiple protocol handlers to participate in a shared message space by forming a rich protocol stack.

6. Reconfiguration

Self-Contained Reconfiguration

The naive reconfiguration is complex and difficult to implement in a system where each meta-object refers directly to each other and requires each other's existence for communication. Using this model, reconfiguration is performed in a self-contained manner (Fig. 6). The steps required to replace meta-object O by its successor meta-object O' without the other meta-objects recognizing it are as follows:

State preservation by posting a message: Meta-object O creates a message that contains its internal state. The preserved state in the message has enough information for the restoration of the meta-object. Once the message is created, meta-object O submits it to the space.

1. Quit: meta-object A finished its work and hence leaves from the space.

2. Restore based on the message: The new meta-object O' withdraws the stored message and initializes its internal state based on it. Once the initialization succeeds, successor O' starts to work.

No negotiation is required between the meta-objects before reconfiguration. Thus, reconfiguration is done in a self-contained manner. This not only simplifies the reconfiguration, but also prevents deadlock and low-level synchronization programming.

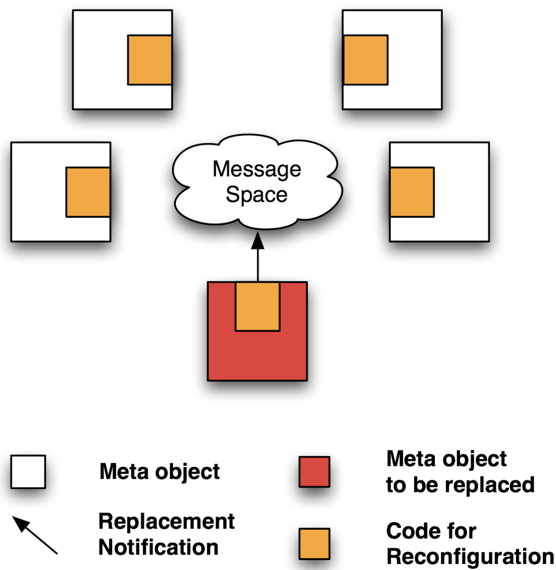

Fig. 6 Self-contained Reconfiguration with Generative Communication

The Unification of State Preservation and Meta-object Communication

Traditionally, state preservation makes use of external storage, such as a file system or a data structure in memory[3][4]. MoleNE[3] introduced the Memento design pattern[5] for this. Therefore, state preservation and restoration were special procedures in reconfiguration. On the other hand, self-contained reconfiguration is accomplished using communication from the dead meta-object O to the future meta-object O', where the message holds the meta-object's state.

Moreover, since generative communication imposes no direct knowledge for a successor object, this model makes it easy to implement other reconfiguration patterns, such as integration of several meta-objects into a single meta-object or the separation of a single meta-object into multiple meta-objects.

This leads to the interesting fact that this model unifies meta-object communication and state preservation.

Safe Reconfiguration

Self-contained reconfiguration greatly simplifies the process of reconfiguration, in contrast to the naive approach of the two-phase commit protocol. Therefore, the new object model helps to satisfy the consistency rules, especially those dealing with state consistency, which help keep the object's state correct after reconfiguration. Operation consistency requires upwards compatibility of a reconfigurable object. Given the flexibility of the model, new meta-objects can be attached. This makes it easier to ensure upwards compatibility. Protocol handlers support referential consistency by encapsulating internal structure. Following all the consistency rules, the proposed object model greatly facilitates safe reconfiguration.

Related Work

Adaptive Middleware Frameworks

Pierre[6] proposed a similar adaptable middleware framework in which objects could reconfigure themselves according to their surrounding environment. However, a single object in Pierre's paper consists of multiple meta-objects. A simple container manages these meta-objects, and thus, as the author noted, this might lead to an incorrect reconfiguration.

MoleNE[3] is another adaptive middleware framework focusing on the wireless environment. MoleNE successfully puts the reconfiguration control of objects to automatons that describe the reconfiguration process.

However, MoleNE's reconfiguration ability is limited for functional components but not for non-functional ones.

Conclusion

We showed that our previous model had 2 drawbacks: 1) fixed configurations and 2) incorrect reconfigurations. Rules that define state, operational, and referential consistency for correct reconfiguration were presented. A naive solution that follows the rules was also presented and found to be too difficult to implement. We showed that the main cause of these problems was the tight dependencies among the meta-objects. Therefore, we introduced loosely coupled communication for concurrent communicating meta-objects in a reconfigurable object. Due to the generality found among loosely coupled communication semantics, we chose generative communication. Generative communication untangles the dependencies among the meta-objects by its indirect, asynchronous, and content-addressing communication semantics using generated messages. With the introduction of generative communication, the proposed object model provides the following features:

- Flexibility and Power: Our model allows a variety of configurations. The reconfiguration of integration and decomposition of meta-objects can also be supported.

- Self-contained Reconfiguration: The object model could reconfigure itself in a self-contained manner, which means that no complex negotiations among the meta-objects are required beforehand.

- Safety: The model follows the three consistency rules for a safe reconfiguration.

- Unification: State preservation during reconfiguration occurs as communication between the predecessor meta-object and successor meta-object.

- Separation of Concerns (SoC): the model can introduce a meta-object for each new concern.

- Novelty and Efficiency: Inter-meta-object communication and reconfiguration of an object are novel applications of generative communication. By specializing generative communication for internal object communication, a highly efficient communication kernel could be made possible.

Based on this model, we implemented a prototype framework called Juice 2[7]. So far, the communication kernel was successfully implemented using standard Java with no virtual machine modification or native library use.

Acknowledgements

This research was partially supported by the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Grant-in-Aid for Young Scientists (B), 17700070, 2006.

References

[1] K. Oda, T. Shin'ichi, Y. Takaichi, The Flying Object for an Open Distributed Environment, Proceedings of the 15th IEEE International Conference on Information Networking (ICON-15), 2001, pp.87-92.

[2] D. Gelernter, Generative Communication in Linda, ACM Trans. Prog. Lang. Syst., 1985, Vol. 7, No. 1, pp. 80-112.

[3] M.T. Segarra, F. Andre, A Framework for Dynamic Adaptation in Wireless Environments, Technology of Object-Oriented Languages and Systems (TOOLS 33), 2000.

[4] I. Warren, I. Sommerville, A Model for Dynamic Configuration which Preserves Application Integrity, Proceedings of the 3rd International Conference on Configurable Distributed Systems (ICCDS '96), 1996, pp. 81-88.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley, Reading Mass. 1995.

[6] P.C. David, T. Ledoux, An Infrastructure for Adaptable Middleware, R. Meersam and Z. Tari, Editors, Springer-Verlag, Lecture Notes in Computer Science, vol. 2519, Proceedings of the Distributed Objects and Applications 2002 (DOA 2002), 2002, pp. 773-790.

[7] K. Oda, H. Najima, Y. Yasutake, T. Yoshida, A Simple, Safe Reconfigurable Object Model with Loosely-Coupled Communication, Proceedings of the IEEE 20th International Conference on Advanced Information Networking and Applications (AINA 2006), 2006, pp.406-411.

Kentaro Oda received the B.S. and M.S. from the Department of Artificial Intelligence, Kyushu Institute of Technology, Japan, in 1999, 2001 respectively. He has been an assistant professor of the Program of Creation Informatics at Kyushu Institute of Technology since 2004. His current research interests include adaptive middleware architecture, multi-agent systems (robotics soccer RoboCup), and distributed systems. He is a member of the ACM, IEEE (IEEE Computer Society).

Shinobu Izumi received the B.S. and M.S. from the Department of Artificial Intelligence, Kyushu Institute of Technology, Japan, in 2004, 2006 respectively. Since April 2006, he has been a PhD student at Kyushu Institute of Technology. His current research interests include disabled access GIS, peer to peer networking and distributed system.

Yoshihiro Yasutake received the B.S. and M.S. from the Department of Artificial Intelligence, Kyushu Institute of Technology, Japan, in 2000, 2002 respectively. He has been an assistant professor of the Department of Intelligent Informatics at Kyushu Sangyo University since 2005. His current research interests include reliable distributed systems, adaptive middleware architecture. He is a member of the IPSJ (Information Processing Society of Japan).

Takaichi Yoshida received the B.S. degree in electrical engineering form Keio University, Japan, in 1982, and the M.S. and Ph.D degree in computer science form Keio University in 1984 and 1987, respectively.
Since 1987, he has been at Kyushu Institute of Technology, and currently is a professor in the Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology. His research interests include distributed computing and object-oriented computing.