Reflection and Concurrency

a position paper for the OOPSLA '92 Workshop on Next Generation Languages

Jeff McAffer

Department of Information Science, The University of Tokyo
and
Object Technology International, Inc.
jeff@is.s.u-tokyo.ac.jp

Introduction

We have worked on and with the Actra [Thomas *et al* 86] concurrent Smalltalk system for the past seven years at Carleton University, the Defence Research Establishment Ottawa and Object Technology International. This work covered most aspects of the language and system design and implementation including virtual machines, integration with real-time operating systems, tasking model, base data type classes rationalization and the design of GUI frameworks. We used the developed technologies in large and small programming teams on real-time embedded applications [Barry 89], turn-key business applications, software development tools and experimental algorithm development projects [McAffer 90a, 90b]. Through these experiences we see several areas in need of improvement. Here we will address only reflection and concurrency.

Recently in the comp.object newsgroup on Usenet, a discussion regarding the design of a new OOP language elicited many varied responses. A large number of the respondents indicated support for reflection and concurrency as being essential in any new language. Our experience agrees with this and here we present a mechanism to support variably asynchronous communications with computational reflection.

From a theoretical viewpoint, the optimal solution would be to have a fully reflective language in which all manner of communications mechanisms, sequential and concurrent, could be seamlessly implemented. In practice this may not be possible. For the mechanisms we propose, we need only have objects which are reflective in the domain of concurrent object intercommunication. Of course, they may be reflective in other ways but that is independent of our desired features.

The language syntax is somewhat independent as well although it should be consistent and logical across all messaging operations. Language constructs may mandate a differentiation between synchronous and asynchronous message passing, concurrent and sequential message sends, parameter passing techniques, etc. This will not affect the presented mechanism as it supports all of these. Our focus is on the design of flexible communication mechanisms making all language constructs and implementation architectures possible and interoperable.

Actra goes part way in that it includes a model of concurrency which is fully integrated with the base language and the underlying OS but it is limited by the host language and architecture. For example, it provides only synchronous message passing. This is a

good choice given the platform and intended problem domains but is not a general solution. Of course, asynchronous communication can be implemented with synchronous messaging and vice versa but that is not satisfactory. Using that technique, the behaviour of concurrent objects is not explicitly described in a form which can be externally analyzed and manipulated.

Desired features

The essential goal it to provide flexible and powerful high-level abstractions of interobject communication which can be processed and manipulated.

This goal can be satisfied by including reflection in the language design and developing a suitable concurrency abstraction. Models of reflection have been proposed by several groups. We have elected to use the ABCL/R model [Watanabe and Yonezawa 88] as the basis for concept and terminology. However, other models of reflection could be used to equal effect. In this short paper we assume that the reader is familiar with the basic concepts of reflection and the notion of meta-objects as defined in ABCL/R and give only a very brief description.

Each concurrent object O has a meta-object $\uparrow$O which defines the concurrent behaviour of O. O is the *denotation* of $\uparrow$O. In a fully reflective system, $\uparrow$O might also define the sequential behaviour of O. Of course, since $\uparrow$O is also an object, it also has a meta-object $\uparrow\uparrow$O and so on. For two objects, A and B to interact, A invokes it message sending mechanism as defined by $\uparrow$A. The resultant message is then transferred to B which invokes its message reception mechanism as defined in $\uparrow$B. Meta-objects can be concurrently accessed and modified thus changing the behaviour of their denotations dynamically.

For a communications abstraction we favour *variably asynchronous messaging*. VAM first came about in our work on distributed simulation where it forms the basis of the Unified Distributed Simulation (UDS) algorithm [McAffer 90a, 90b]. The messaging scheme is not limited to the simulation domain where time provides an inherent sequencer but rather, VAM is applicable wherever a valid sequencer can be found. Similar work has been done at Mitre [Bensley, *et al* 88] where they developed a message-stamping scheme allowing generic object-oriented programs (i.e., not simulations) to be run using Time Warp (see below) messaging. VAM can be used in an analogous way to effect variably asynchronous behaviour for arbitrary objects.

By default all objects are concurrent until they interact. The nature of their interaction defines the nature of their concurrency. Objects using VAM can dynamically and reflectively adjust how and when messages are accepted (freeing the sender to continue), processed and dispatched thus changing their concurrent behaviour. Note that to accept a message, is to guarantee to process it in the correct sequential order relative to the other messages in the system.

VAM is an extension of the communications mechanism found in the Time Warp [Jefferson 85] simulation system. In Time Warp, simulation objects interact asynchronously until a temporal inconsistency is detected. That is, until an object receives a message from its *past* (in simulation time). To resynchronize, the object rolls back erroneous computations and message sends, and proceeds with processing the new message. The algorithm is guaranteed to progress and, using techniques such as lazy evaluation and lazy cancellation, is quite efficient on a large number of

problems. Time Warp explicitly supports only this asynchronous mode of communication.

Other simulation algorithms (e.g., Chandy-Misra [Misra 86]) are completely synchronous. No message processing can take place until the message's sender and receiver are synchronized in simulation time. As a result, the simulation can only proceed as fast as the slowest object. Asynchronous communication cannot be modeled efficiently.

Through reflection, VAM allows interacting objects to *decide* how messages should be handled (i.e., sent, received and processed) and vary their own communication behaviour along a continuous spectrum from synchronous to asynchronous.

A simple ABCL/R implementation of Time Warp in which the entire mechanism is implemented by redefining, in the meta-object, the way an object receives messages is presented in [Watanabe and Yonezawa 88]. VAM extends this by defining two parameters, *risk* and *aggressiveness* for each object. Risk is how far into the future an object can send a message while aggressiveness is from how far into the future an object is willing to accept a message. This idea has analogies in speculative computing with risk being how much speculation an object can add to the global computation and aggressiveness being how much of the speculation in the system an object is willing to use.

The control of these two parameters (and others) by the meta-object, while typically based on the denotation's state and history, can be determined in any application specific way. Asynchrony increases as risk and aggressiveness increase. Setting both to ∞ for all objects makes a completely asynchronous system. Conversely, setting all to zero makes a completely synchronous system. Settings between zero and ∞ effect partially asynchronous behaviour.

Independent of an object's code, its aggressiveness, for example, can be changed to allow messages to be accepted (thus freeing the sender) but held over at the receiver for furture processing. In a synchronous system a sender must wait until the receiver replies. In an asynchronous system senders never block. In VAM, how long senders must wait for message acceptance is inversely proportional to the receiver's aggressiveness.

Such facilities are useful wherever objects have dynamically changing communication needs. Consider a traffic system model containing highways, city streets, intersections, crosswalks, and cars. On highways, cars tend to be asynchronous while on city streets, they must synchronize with intersections, crosswalks and slow moving cars. As a car moves from a street to a highway, it should adapt to the new concurrent environment. If it does not, unnecessary synchronization will be forced and efficiency reduced. Moving from a highway to a street will result in a large number of rollbacks if the car does not reduce its asynchrony. This adaptation should be done at the meta-level and should not require motifications to the car model itself.

The reader is referred to [McAffer 90a] for a full discussion of this messaging scheme, a proof of sequencer relative consistency and the implementation of UDS (and VAM) in Actra on a multiprocessor MC680x0 computer running the Harmony Real-Time Operating System [Gentleman 85].

Ramifications

VAM and reflection as discussed here have bearing on a number of issues in language and system design.

Encapsulation - The concurrent behaviour of objects can be held private or published by the object itself. Typical systems use some sort of central kernel code to effect message passing, thus the message passing behaviour of all objects is identical and therefore public. Since each object is in control of its own concurrency, none of its internal details must be made public to facilitate control.

Reflection - Adding reflection as proposed enables high-level abstractions or *first class* representations of concurrency concepts in the system. We would like to see more general support for reflection within new languages but do not require it here.

Integration, portability and interoperability - The inclusion of reflection and VAM allows objects with vastly different concurrent behaviours to communicate. This is fundamentally important to integration, portability and interoperability. The use of reflection allows us to build a communication abstraction. This in turn, distances users from the underlying realities of the system and increases portability, and interoperability.

Concurrency - Concurrency at a low-level is too hard to deal with. High-level abstractions are sometimes too high, restricting the user. Reflective communication facilities allow the user as much or as little control as they want by giving them the ability to modify the abstraction dynamically. For example, when objects recognize repeated synchronization patterns they can attempt to form more efficient communication links or groupings.

Extensibility - New communications constructs (e.g., barrier synchronization and groups) can be added cleanly and consistently.

Distribution - Having a high-level model of interaction enables objects to better adapt to changing environments. In a distributed system this enables features such as self-scheduling, self migration for load balancing and participation in object groups.

Productivity - Distributed and concurrent systems are notoriously hard to understand and debug. Having open communications mechanisms gives designers and programmers access to object interactions and admits the implementation of tools for communication tracing and debugging and performance monitoring.

Limitations and remaining problems

Most of the difficulties in implementing these features are with respect to efficiency. For example, the Mitre time-stamping scheme is not efficient. Since time-stamping is only used to provide a message sequencer, any valid sequencer will allow VAM to guarantee sequential consistency of object interactions. In many cases, it may be possible to provide application derived sequencers efficiently. Finding such sequencers however, may be difficult. Languages using VAM should provide a number of *standard* sequencers which characterize typical systems.

The implementation of reflection and VAM may require more message passing than traditional systems. This could pose efficiency problems on some architectures.

As with the sequencer, determining how and when to set the risk and aggressiveness may be quite difficult. Can generic methods be developed for general cases?

We have not shown that this model can be used to easily describe all desired behaviours (i.e., sequencers and message patterns).

VAM requires a rollback mechanism to guarantee sequential consistency. It will be challenging to provide and efficient yet general rollback facilities.

Ultimately, having such a degree of flexibility is a two edged sword. Mechanisms without guidelines (or at least hints) on how to use them are not very easy to use. We should strive to provide as many of the typical interaction behaviours as possible and guidelines for extending the base communication and control patterns.

Conclusions

Most of the ideas presented here are not new. They have been developed in other domains and independently of one another. This paper attempts to combine them to improve future languages. We propose that the inclusion of reflection at the level of interobject communication is essential and that the implementation of a scheme like VAM is a logical consequence of reflection. Their implementation, at the right level in the system, will enable a wide (perhaps infinite) range of concurrent behaviours on many architectures. The affects of this range from portability and interoperability to support of distribution and sophisticated concurrency control.

References

Barry, B., (1989) "Prototyping a Real-Time Embedded System in Smalltalk", in *Proceedings of OOPSLA '89*, New Orleans, LA, pp. 255-265, September.

Bensley, E.H., *et al* (1989), "An Execution Model for Distributed Object-Oriented Computation", in *Proceedings of OOPSLA '88*, San Diego, CA, pp. 316-322, September.

Gentleman, M. (1985), "Using the Harmony Operating System", ERB-966, NRCC No. 24685, National Research Council of Canada, Ottawa, Ontario.

Jefferson, D., (1985), "Virtual Time", *ACM Transactions on Programming Languages and Systems*, 7(3), July.

McAffer, J. (1990a), "Unified Distributed Simulation", M.C.S. Thesis, School of Computer Science, Carleton University, Ottawa, Ontario, March.

McAffer, J. (1990b), "A Unified Distributed Simulation System", in *Proceedings of the 1990 Winter Simulation Conference*, pp. 415-422, New Orleans, LA, December.

Misra, J. (1986), *Distributed Discrete-Event Simulation*, Computing Surveys, 18(1), pp. 36-65, March.

Thomas, D., *et al* (1986) *Actra: A Multitasking/Multiprocessing Smalltalk*, SCS-TR-92, School of Computer Science, Carleton University, Ottawa, Canada, May.

Watanabe, T. and Yonezawa, A. (1988), "Reflection in an Object-Oriented Concurrent Language", in *Proceedings of OOPSLA '88*, San Diego, CA, pp. 306-315, September.