Comprehensive Concurrency Controls Classification: Achieving Reflection in Concurrent Object-Oriented Systems

Tzilla Elrad, Ufuk Verun

Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616

cselrad@minna.acc.iit.edu   verun@iitmax.acc.iit.edu

As stated in the OOPSLA/ECOOP'90 Report on the Reflection and Metalevel Architectures in Object-Oriented Programming "Work on reflection in concurrent systems is still in its infancy, but it seems that concurrent systems can gain the most from reflection". In this position paper we would like to propose the use of the Comprehensive Concurrency Controls [Elrad] to help in

A. the classification of various elements of possible concurrency controls. (messages, massage queues, concurrency, synchronization, scheduling etc)
B. the expressiveness of those controls needed to achieve causal connectivity between reactive/adaptive intelligent systems and their domain.
C. the architecture of OO languages that support A and B.

We would like to address a position which relate to the following three requirements.

* The exhaustive requirement: Should synchronization and concurrency controls requirements be expressible within the language? What aspects should be expressible?
This requirement measures the degree of reflection expressible by the language.

* The dynamic requirement: Should concurrency controls be dynamically modified?
This requirement measures the degree of causal connectivity expressible by the language.

* The congruity requirement: Should such a language support an external handler for concurrency and synchronization controls the same way it supports external handlers for domain information?
This requirement measures the degree of uniformity with respect to the object orientation.

To help us better understand and address these issues we propose the use of the COMPREHENSIVE SCHEDULING CONTROLS proposed in [Elrad].

The Comprehensive Scheduling Controls

In most reactive/adaptive real-time systems some control over indeterminate behavior constructs is needed to realize the overall system behavior. The set of all possible concurrency and scheduling controls used by a language is termed the Comprehensive Scheduling Controls. The comprehensive concurrency controls compute what computation could/should be done next and hence provide linguistic mechanisms for achieving reflection.

The Comprehensive Scheduling Controls are grouped into two classes: Availability Controls and Race Controls.

Availability Controls are used to manipulate the state information about a system to determine which actions could be performed in the next step of execution. Changes in the system state can be reflected in the system behavior by utilizing the Availability Controls. The causal connection between system domain and internal structures that control system decision among multiple futures can be established and controlled by utilizing the Race Controls.

```
              Comprehensive Scheduling Controls
                                 |
                     ---------------------------
                     |                         |
       Availability Controls             Race Controls
                     |                         |
        Consensus Control                 Priority Control
        Private Control                   Preference Control
        Mutual Control                    Forerunner Control
```

Availability Controls

Availability Controls are those controls which enable or disable a non-deterministic choice for selection within a selective wait construct. In a reactive/adaptive system, the event-driven specification defines the allowable state transitions from the current state; guaranteeing that the system will be restricted to these allowable transitions is an important ability for a language. Availability Controls also provide the flexibility to control these alternatives dynamically, which allows the language to accommodate the constantly changing system mode, so they provide an implementation of causal connectivity. Availability of any of the alternatives for selection is derived from certain constraints applied to each alternative, such as boolean expressions, communication readiness or any combination of such constraints.

Availability Controls are classified as Consensus Control, Private Control, Mutual Control to reflect the different constraints that may be applied to an alternative to control behavior.

Consensus Control.

Consensus Control represents the capacity to enable or disable a non-deterministic choice for selection based on pending communication requests originating outside the local environment of the task. The consensus control would particularly enable a specific channeling of a communication to a response. By using consensus control a task may respond to an event which is originated outside the task.

Private Control.

Private Control represents the capacity to enable or disable a nondeterministic choice for selection based on a boolean expression whose value is determined according to the local state of the task. State transitions may be conditional, and the behavior must be sensitive to these fluctuating conditions.

Mutual Control.

The capacity to enable or disable an alternative for selection based on a boolean expression whose value is determined according to both the local state of the task and parameters passed in by the caller task. Mutual control enables a task to check the values of the parameters passed in by the caller task and use this information to either accept the call, or to suspend it.

Race Controls

During the execution of a concurrent program, tasks are racing to be scheduled when a resource becomes free at the program level, open alternatives are racing to be selected when an alternative construct is executed at the task level or entry calls are racing to be accepted to initiate an inter-task communication at the entry level. An event must be selected from the available events. Control mechanisms become necessary to resolve race conditions among system entities. Race controls are those controls which resolve a race for either task scheduling, alternative selection or inter-task communication. Race Controls compute (by that possibly affect) a system’s own prioritization.

Race Controls are classified according to the control levels as in the following.

Priority Control.

Priority Control is the mechanism to resolve races in program level, among the tasks that are eligible for execution. For example, in a reactive/adaptive environment, safety goals and normal operating goals should be assigned different priorities.

Preference Control.

Preference Control is the mechanism to resolve races in task level, within a nondeterministic select construct, among eligible alternatives. In E&D (Explicit and Dynamic) specifications, explicit preferences are specified within an alternative construct for all alternatives. Use of variable slots for preferences will provide a dynamic Preference Control mechanism. Allowing the update of preferences within a function body will improve the flexibility and will eliminate some limitations that can be caused by the global task scheduler module. In addition to task scheduler being able to update these preferences, tasks themselves within an accept body can alter the preferences of their alternatives and some other tasks' alternatives. This requires an explicit specification of preference variables globally so that every task has access to every preference variable. The accesses are further controlled by the use of a privilege assignments to the tasks.

Forerunner Control.

Forerunner Control is the mechanism to resolve races in entry level, among pending communication requests in an entry queue. An explicit queuing strategy is specified to handle the communication requests arriving at the entry. The queuing strategy can be either a predefined rule like FIFO, or some user declared strategy. By allowing the specification of user defined strategies, the language will not be restricted to a small set of rules. The user defined strategies can be declared in special modules out of the program, allowing the user to build libraries of scheduling rules. In addition to adding an explicit queuing strategy specification to each entry to provide Forerunner control.

In the early stages of the concurrent object-oriented languages, the problem of encapsulating data and operations has been recognized and solutions have been provided. The idea of encapsulating the concurrency and scheduling requirements the same way as data and operations has not emerged for a while. Some languages, like Ada [Ada] and Concurrent C++ [Gehani], provided specification parts where data and operations can be encapsulated and re-used. It was the advances in the concurrent object-oriented language design that gave some ideas about how beneficial it would be to treat concurrency and scheduling decisions as data and operations.

There are a number of object-based and object-oriented languages that try to incorporate synchronization control as part of specification parts [Tomlinson, Gehani, Ada]. For example, message-based enabled-sets mechanism was introduced on top of the actor model [Tomlinson, Agha 1986], protected record mechanism was introduced as part of the Ada 9X revision process [Ada], Capsules was introduced to provide an encapsulated synchronization and scheduling control in Concurrent C++ [Gehani]. These improvements were not easy because of the conflicts between inheritance and synchronization code. In a message-based environment like actors, code for synchronization sharing and the logic of the application were affecting each other [Briot and Yonezawa 1987]. The modifications in the synchronization code usually necessitate cascaded modifications through the inheritance hierarchy [Kafura and Lee 1988, Decouchant et al. 1988].

Our research has been mainly towards better languages for the realization of reactive, adaptive soft real-time systems. These are the systems that have internal states, interact with their environments, react to external events and adapt their behaviors if necessary under soft real-time constraints. The languages for such systems should incorporate a comprehensive set of synchronization and scheduling controls, so that the designers will have the absolute control over their systems which do not have any tolerance against run-time inconsistencies. For a long time, these set of controls have been implemented as parts of the underlying run time systems, have been mixed with the actual code that performs the required operations or have been provided as annexes to the programming languages. The resulting systems were highly dependent of the run time systems and were hard to integrate with some other existing systems. Porting programs was a mess and migrating applications was almost out of question. A strong discipline between system information and domain information can be of a great benefit to those systems.

Reactive, adaptive systems are implemented by gluing together some reflective modules that perform specific system tasks. External handlers are syntactically recognizable entities that can be dynamically modified from outside the module. These handlers are templates to instantiate and modify any specific dimension, such as structure, functionality, behavior, concurrency controls or synchronization and hence support the concept of computational reflection [Maes]. They provide means to control the ontology of the modules, and to meet the computational reflection imposed by the various system modes. Execution schemes are dynamic, changeable, adaptable and extensible.

A multi-dimension building block is a structure which is capable of encapsulating every aspect of the requirements of a module's specifications so that, for example, scheduling, concurrency and synchronization could be expressible. Explicit concurrency controls are languages contracts to enable a high degree of reflection. Providing explicit and dynamic controls enable better causal connectivity. Moreover, each one of those dimensions could be controlled by an external handler (most OO languages allow only for the manipulation of data and operations from outside the module). Integrating the block for multiple uses, either in the same system or for later reuse, is facilitated by these external handlers.

References

[Ada] Ada 9X Mapping Document, Volume I, Mapping Rationale, Office of the Under Secretary of Defense for Acquisition, Department of Defense, Washington, D.C., March, 1992.
Ada 9X Mapping Document, Volume II, Mapping Specification, Annexes, Office of the Under Secretary of Defense for Acquisition, Department of Defense, Washington, D.C., March, 1992.

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

[Briot and Yonezawa 1987] Briot J.P. and Yonezawa A., Inheritance and Synchronization in Concurrent OOP. Proceedings of ECOOP '87, Springer Verlag LNCS 276, 1987, pp. 32-40.

[Burns] Burns A., Wellings A.J., In Support of the Ada 9X Real-Time Facilities, ACM Ada Letters, January/February 1992, Vol.12, No.1, pp. 53-64.

[Elrad] Elrad T., "Comprehensive Race Controls: A Versatile Scheduling Mechanism for Real-TimeApplications". Proceedings of the ADA Europe Conference, ADA The Design Choice, Ed. Angel Alvarez, Cambridge University Press, June, 1989.
T. Elrad, Final Report, CECOM, Center for Software Engineering Advanced Software Technology, CIN:C08092KU 000100, February, 1990.

[Gehani] Gehani N., Capsules: A Shared Memory Access Mechanism for Concurrent C/C++, AT&T Bell Laboratories, 1992.

[Maes] Maes P., Concepts and Experiments in Computational Reflection, Proceedings OOPSLA 1987, pp. 147-155.

[Tomlinson] Tomlinson C. and Singh V., Inheritance and Synchronization with Enabled-Sets, OOPSLA 1989 Proceedings, pp. 103-112.

[Kafura and Lee 1988] Kafura D. and Lee K., Inheritance in Actor Based Concurrent Object-Oriented Languages. TR 88-53, Dept. Computer Science, Virginia Polytechnic Institute and State University, 1988.

[Decouchant et al. 1989] Decouchant D., Krakowiak S., Meysembourg M., Riveill M., Rousset de Pina X., A Synchronization Mechanism for Typed Objects in a Distributed System. Proceedings of the ACM SIGPLAN

Workshop on Object-Based Concurrent Programming, SIGPLAN Notices, 24:4, pp. 105-107.