Reflective Controls for Intelligent Distributed Objects

En-Hsin Huang                                Tzilla Elrad
Lucent Technologies, Inc. Illinois Institute of Technology
Naperville, IL 60566, USA    Chicago, IL 60616, USA
enhuang@lucent.com            elrad@iit.edu

KEYWORDS
Autonomous, reflective controls, distributed objects, intelligence.

ABSTRACT
A reflective decision control mechanism is presented for dealing with client request scheduling issues in distributed environment. We propose a component-based reflective architecture that incorporates an intelligent meta agent for assisting distributed objects in decision making. This configuration enables autonomous entities to become more responsive in a robust environment. We use a generic wireless communication application as an example to illustrate how to equip objects with reflective scheduling capability for processing cellular calls. Such capability is critical to the successful implementation of distributed software that possesses intelligence, flexibility and adaptation capability.

1 INTRODUCTION
Reflective systems allow computations to examine and modify properties of their own behavior with consideration for information from the external environment [8,13,16,19]. These systems are examples of meta-level architectures; software architectures which operate not only on a base level of normal computation, but also on a meta-level of computation about the system's operation. When distributed objects are associated with reflective agents, certain level of intelligence and adaptation capability can be implemented. This arrangement makes distributed objects software capable of managing request conflicts and schedule concurrent access. Object systems that are based on the reflective architecture provide a promising solution [2,8,16].

As an example, let us examine a generic wireless communication application. To process a cellular call, a set of distributed objects have to work in a cooperative fashion to manage task such as request registration, call distribution, and call processing. Before a request is handed over to the call processing agents, the call assignment agent has to perform call registration and load-balancing. A non-reflective agent which lacks the capability to fully interact with its environment in managing stimuli and feedback will surely fall short of achieving the optimal call distribution. On the other hand, objects with reflective decision control capabilities can make intelligent decisions based on the work load situation and other environmental factors of each processing agent.

Section 2 briefly examines the dynamics of the concurrency aspects of complex systems. We specifically address the issue involving the ability to enable or disable non-deterministic choice for selection within a servicing entity. Section 3 presents an architectural design that incorporates reflective agents for managing request scheduling. It also serves as a framework to better understand how concurrency controls are to be properly integrated into object languages. Section 4 illustrates how to formulate and employ reflective control strategies for resolving selection conflicts. Section 5 compares our approach to other related works.

2 SCHEDULING CONTROLS
In a distributed environment, autonomous objects constantly interact to achieve coordination and information exchange. Messages may arrive at any given moment. A selection conflict occurs within a server entity when multiple conflicting requests are competing to be scheduled. Scheduling Controls [5,9] decide what should be done next among available choices, and hence support scheduling.

Two forms of message race conflict are considered here. One, also known as the inter-method race, is concerned with resolving conflicts among executable method entries inside an object. Since requests may arrive for different services, an intelligent scheduling scheme must determine the best choice to meet system requirements.

Section 2 briefly examines the dynamics of the concurrency aspects of complex systems. We specifically address the issue involving the ability to enable or disable non-deterministic choice for selection within a servicing entity. Section 3 presents an architectural design that incorporates reflective agents for managing request scheduling. It also serves as a framework to better understand how concurrency controls are to be properly integrated into object languages. Section 4 illustrates how to formulate and employ reflective control strategies for resolving selection conflicts. Section 5 compares our approach to other related works.

2 SCHEDULING CONTROLS
In a distributed environment, autonomous objects constantly interact to achieve coordination and information exchange. Messages may arrive at any given moment. A selection conflict occurs within a server entity when multiple conflicting requests are competing to be scheduled. Scheduling Controls [5,9] decide what should be done next among available choices, and hence support scheduling.

Two forms of message race conflict are considered here. One, also known as the inter-method race, is concerned with resolving conflicts among executable method entries inside an object. Since requests may arrive for different services, an intelligent scheduling scheme must determine the best choice to meet system requirements.
One common solution is to treat alternatives with equal preference and solve the message race arbitrarily. This approach simplifies the scheduling job, but fails to express the relative preference relationships among alternatives. An alternative is to enforce preference relationships either 1) implicitly based on the textual layouts as in [1] or 2) explicitly by providing linguistic supports for expressing relative preference relationships among execution paths as in [3] and [14].

The second form of message race conflict is known as the intra-method race which is concerned with requests racing for the same service. It aims to manage request selection over waiting calls in an entry queue. A common solution is to have the underlying run-time system provide the default scheduling mechanism. This arrangement makes object software unpredictable. One alternative is to have selection condition expressed in terms of message contents to support formulation of a more responsive scheduling mechanism as observed in [1] and [3].

3. INTELLIGENT OBJECTS
An autonomous distributed object is modeled as an event-driven encapsulating entity with a base object and a proxy meta agent. This arrangement promotes the principle of separation of concerns since each level deals with only a specific functionality of the object entity. The base level models the structural and behavioral aspects of its assigned object. The meta level deals with the behavioral aspect of an object. It manages message queues and performs object state monitoring. It is responsible for scheduling client requests and maintaining the integrity of the shared resources.

3.1 The Base Level Component
A base level object definition consists of data specification and method implementation as in traditional object-oriented languages. We choose C++[17] to be the default base-level language mechanism since C++ is widely used and it can readily support the above sub-component definitions.

3.2 The Meta Level Component
A metaagent serves two purposes. First, it can be viewed as an extended interface to the base-level object class. A metaagent permits the incorporation of user-specified requirements to the base-level object. These requirements cover synchronization conditions, real-time constraints, and priority ordering of requests. The metaagent definition has three major components. They are:

- the synchronization controls depository enables the specification of acceptance conditions for client requests;
- the race controls depository enables specification of scheduling policies for managing inter- and intra-method race; and
- the event flow specification section binds individual method with various functionality (e.g., decision controls, post-action triggers) and specifies what method(s) can be executed concurrently.

Secondly, the meta-object serves to monitor and regulate the behavior of the associated object at run time base on some pre-determined requirements. The event scheduler component of the meta agent takes control over how client requests are to be received and processed. By default, requests are accepted unconditionally and executed in mutual exclusion. Calls to the same method are processed in First-In-First-Out order.

When a more intelligent scheduling capability is required, other functional sub-components are consulted. The synchronization controller is first called upon to manage conditional acceptance of client messages. Next, the parallelism controller determines which open methods can proceed in parallel. The race controller will then resolve possible inter- and intra-method race conflicts. Finally, the event scheduler schedules the selected method(s) for execution and triggers post-action (if any) to reflect changes in object state. In this paper, we will focus more on the race controller and how to resolve message scheduling conflicts.

When the inter-method race occurs, the race controller must select the best choice that meets certain specific requirements. The appropriate policy (defined in the policy depository section) is activated to provide a preferential ordering to all method entry alternatives. In this way, the race controller can be more reactive and dynamic at run-time in managing request calls.

When the intra-method race occurs, the race controller will be confronted with the problem of how to manage requests within the message queue. The appropriate policy (defined in the policy depository section) is invoked to indicate how to sort and select requests. Information related to object state and the optional input values can also be included in reaching the final scheduling decision.

3.3 Reflective Scheduling Control Mechanisms
In our design, the default root of all meta objects is class AGENT. Each base class has a companion meta class (either the default root class or user-defined one). The default meta class provides very little scheduling support. Methods inside the base class are executed in mutual exclusion fashion. Furthermore, message scheduling conflict is resolved based on requests' arrival order in the queue. To implement a more dynamic scheduling mechanism, new meta class(es) can be derived from the root class. Inheritance mechanism is adopted for defining new classes based on existing classes with some localized modification or extensions.
Two separate sets of class hierarchy are maintained (e.g., one for base class and the other for meta class). The proposed meta-class constructs permit formulation of adaptive decision controls for managing inter- and intra-method race conflicts. Besides supporting code reuse, these constructs also support formulation of reusable decision controls.

4 MANAGING SCHEDULING CONFLICTS

We can program the call processing agent of the generic wireless communication application to process requests. In our example, a distributed call processing agent provides two basic services: one is to enable mobile connection and the other to permit callers to configure customer information on-the-fly. The former service type can be rendered according to some pre-defined service priority. Based on the service plan the caller subscribes and the nature of the call, we can have four levels of priority. They are, ranked from the highest to the lowest, emergency (e.g., the 911 calls), premium, plus, and regular service. Figure 1 provides the base class definition of the call processing agent.

```cpp
// base class
class call_processing {
  protected:
    /* denoting preference level for managing inter-method race*/
    int processCallPref, configInforPref;

    /* update preference variables */
    void updateTurn(int processCallTurn, int configInforTurn)
    {
        processCallPref = processCallTurn;
        configInforPref = configInforTurn;
    }

  public:
    /* initialize preference variables */
    call_processing (void) {
        processCallPref = 1;
        configInforPref = 0;
    }

    void processCall(int serviceLevel);
    void configInfor(long callForwardNum);
};
```

Figure 1. Reflective Intra-Method Scheduling Management

```cpp
// meta class
meta call_processing {
  /* policies for resolving inter- and intra-method race */

  race:
    /* return the values of preference variables for managing inter-method race*/
    inter int processCallPref Level(void) {
        return processCallPref;
    }

    inter int configInforPref Level(void) {
        return configInforPref;
    }

    /* intra-method race policy */
    intra int queueSelection(int serviceLevel) {
        priorityLevel = mapPriority(serviceLevel);
        return priorityLevel;
    }

    /* bind methods to their respective scheduling controls */
    events:
        prefer accept processCall(serviceLevel)
        governed-by: queueSelection(serviceLevel)
        level_of: processCallPref Level()
        post-action: updateTurn(0,1);

    prefer accept configInfor(serviceLevel)
        level_of: configInforPref Level()
        post-action: updateTurn(1,0);
}
```

Figure 2. Reflective Intra-Method Scheduling Management

Figure 2 defines the associated meta agent and binds decision controls with the two public member methods. In the race section, programmers can specify the intra-method race scheduling policy to implement the above scheme. It is done by prefixing policy methods with the modifier intra in the race section. The governed-by clause specifies which intra-method control strategy is to be employed for selecting pending requests. Passed arguments (e.g., message content) from client objects can be used to arbitrate the order by which requests are to be admitted for service.

This capability is especially helpful in devising the intra-method scheduling policy for sorting and selecting requests from the message waiting queue. It allows
prioritization of calls by performing a screening process based on the client information.

In this example, the intra-method policy \textit{queueSelection} (int \textit{serviceLevel}) for method \textit{processCall} incorporates values of the parameter variable \textit{serviceLevel} (bound at run-time) and, thus supports a dynamic scheduling scheme. This feature permits a server entity to take into account both the user’s requirements (e.g., level of urgency) and the current system information (e.g., the overall performance) before reaching any decision. It permits the formulation of a more responsive and intelligent scheduling mechanism.

Note that we can also program the \textit{call process agent} object to take turns in serving requests for processing mobile calls and on-the-fly customer information modification (e.g., changing the call-forwarding number through the mobile handset). The \textit{prefer-of-level} construct permits association of member methods and their respective preference policies. These policies return discrete values denoting the preference level of the associated methods. In the \textit{event} section, users can specify the relative preference relationships among base object methods.

The keyword \textit{prefer} and the clause of \textit{level} explicitly bind race control policies with the appropriate methods. The two variables, \textit{processCallPref} and \textit{configInforPref}, represent the relative preference relationship between the two alternatives \textit{processCall} and \textit{configInfor}. These variables are managed by the newly introduced scheduling policies. Precaution must be taken to ensure that the reordering of preference relationships will not lead to the starvation of other pending requests. The initial set-up (after executing class constructor method) ensures that calls for service \textit{processCall} are given higher priority. After processing a request, the server will prioritize requests for service \textit{configInfor}.

In a non-adaptive scheduling scheme, the two preference variables are initialized bound automatically in the class constructor. Once the value is assigned, no further changes are allowed. In an adaptive scheduling scheme, preference variables can be modified at run-time to reflect changes of system/mission priority (as in Fig 2).

To reflect changes in the relative preference relationships, the post-action method \textit{updateTurn} is invoked after the execution of a method body. This feature makes autonomous distributed software components more configurable in terms of their ability to respond intelligently to feedback. It is permissible to have member methods share the same preference level; all the executable methods are scheduled in the First-In-First-Out order.

5 Comparison with Other Related Works

Only a handful of work [4,14] attempted to provide extended language support for formulating/fine-tuning scheduling strategies in managing indeterminate object behavior. These approaches only address scheduling conflicts that occur when clients make entry calls for different services at the server site. The implementations are based on Ada tasking concurrency model. In this case, message preference is determined based on the textual ordering of entry alternatives in the program context. These attempts resulted in languages that are limited in power and could not support the notion of computational reflection.

Actor model-based reflective system such as [2], [8] and [16], have the role of \textit{meta-object} expanded to permit the management of request scheduling. However, these implementations lack the capability for expressing synchronization schemes involving passed information. They provide very limited support for formulating static preference scheduling strategies. The existing facility disallows the expression of equal preference levels among alternatives. In this case, once the overall relative preference relations for method entries are determined at compile time (based on textual appearance of method entry names), such relations cannot be modified further.

All the above-mentioned approaches have no explicit support for handling \textit{intra-method} race. The scheduling strategy is implicitly embedded within the language semantics. In most cases, requests competing for the same service are processed based on their arrival order. However, it imposes deterministic selection strategy and limits its usefulness for adaptive systems.

6 CONCLUSIONS

A major contribution of our work is to provide a practical framework for applying the \textit{reflective architecture} to supporting dynamic real-time resource management, monitoring, and request scheduling for complex systems. The proposed object design and scheduling control constructs serve to illustrate how to achieve the proper integration of decision controls and object-oriented features for supporting reflective computation. The meta class specification consists of smaller, programmer-specifiable slots for the declarative programming of decision controls.

In this case, synchronization controls are specified and are managed in turn by an object’s \textit{meta agent} for supporting reflection. In addition, users can formulate appropriate scheduling strategies to address concurrency controls at their respective level of concern (as discussed in Section 2). This increases the control programmers have over the responses and behaviors of concurrent adaptive objects. By associating the handlers with
various programmer-specified race control strategies, users can fine-tune the scheduling strategy until an acceptable system response is attained. Currently, we have implemented an extended C++ compiler to permit the association object definition components at the base-object and meta class level. The run-time library provides basic concurrent facilities for formulating thread control, message management, and scheduling control mechanisms.

REFERENCES


Dr. En-Hsin Huang graduated from the Illinois Institute of Technology and currently works as a Wireless software architect at Lucent Technologies. His research interest includes in concurrent and distributed object-oriented languages, software architecture for adaptable and extendible software systems, and object-oriented database systems.

Dr. Tzilla Elrad graduated from the Technion, Israel. Currently she is a research professor at the computer science department, Illinois Institute of Technology. Dr. Elrad interests are in concurrent and distributed programming languages, formal methods for concurrent programs and software architecture for adaptable and extendible software systems.