JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol.6, No.1, January-February 2007

UML-based Approach to Specify Secured, Fine-grained Concurrent Access to Shared Resources

Jagadish Suryadevara, Birla Institute of Technology & Scinec, INDIA
Shyamasundar R.K., Tata Institute of Fundamental Research, INDIA

Abstract

In object oriented paradigm, a concurrent system can be regarded as a collection of autonomous active objects which synchronize and communicate through shared passive objects. In this paper, we propose a UML-based approach to specify secured, fine-grained concurrent access to shared resources ensuring data integrity and security. The goal of the approach is to develop the UML specification with precise executional semantics, yet independent of low-level synchronization primitives and implementation environment. The approach is largely inspired from the language constructs of CDL*. A light-weight extension of UML 2.0 meta-model is proposed for the required constructs and semantics. UML *protocol statemachine* is used to define the access protocol for shared resources and UML *activity* is used to specify the behavior of methods implementing plausibly concurrent operations. The UML *activity* construct is extended to support concurrency features; *synchronization regions, mutual exclusion* and *conditional synchronization* not supported in current UML2.0 semantic model. The approach can be easily extended to a programming framework of design and coding.

1 INTRODUCTION

Object oriented paradigm, due to features like *abstraction*, *data encapsulation* supports design methodologies for various kinds of complex systems like Real-Time, Embedded, Concurrent, Mobile, and Distributed. In particular, the *abstraction* feature supports formal framework for reasoning with the system through static or dynamic analysis. Due to these reasons efforts of OMG, Object Management Group, in promoting and refining UML, Unified Modeling Language [2], are received with wide-spread enthusiasm and adoption. The recent UML2.0 specification with well-defined *semantic foundation* based on fine-grained *action semantics* is the result of continuous ongoing efforts of UML community including formal methods researchers towards semantic refinement of UML

As UML is intended by its designers to be a *family of languages*, its specification left *several semantic variation points* to be defined by the requirements of the domain under

Cite this article as follows: Jagadish S, Shyamasundar R.K.,: "UML based approach to specify secured, fine-grained concurrent acces to shared variables", in *Journal of Object Technology*, vol. 6, no. 1, Januar-February 2007, pp. 107-119 http://www.jot.fm/issues/issue_2007_01/article3

consideration. In addition to these semantic variation points UML provides standard *extension mechanisms* known as *profiles*, *stereotypes, and constraints* to support specific domain modeling

In spite of several efforts, concurrency in UML remains an active research area, requiring concrete approaches to precise modeling to support the programming activity. UML through its active object paradigm, provides various mechanisms to specify concurrency;

- *isActive* meta attribute of metaclass *Class* to specify whether the object is active or passive,
- *concurrency* meta attribute of metaclass *Operation* to specify concurrent operation invocation, and
- orthogonal regions of the state machines to specify concurrent activities in an object.

At run time, these mechanisms should work together correctly in order to ensure the correct behavior of a model. But, unresolved ambiguities and inconsistencies among various mechanisms remain a main hurdle in specifying precise concurrency in UML. Though good designers can avoid these inconsistencies, expressive constructs with well defined semantics are desirable. In this paper we have suggested improvements to the UML semantic model to specify fine grained concurrency. But it is not our goal to define formal semantics in this paper, a task that shall be taken up in future work.

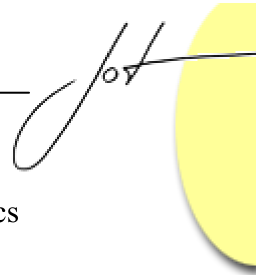
In this paper, we propose a UML-based approach to specify fine grained concurrent access to shared resources independent of implementation aspects. The rest of the paper is organized as follows. Section 2 states the design goals of the proposed approach in the specification of concurrency. Section 3 describes the inadequacies in UML semantics for specifying concurrency using classical *readers-writers problem*. In Section 4, we provide an overview of CDL*. The approach of the paper together with proposed extensions to UML semantics is described in Section 5. Section 6 describes related works.

2 GOALS OF THE APPROACH IN THE SPECIFICATION OF CONCURRENCY

The approach of this paper is largely inspired from the semantics of the language constructs of CDL* [1]. The CDL* constructs provide (i) data integrity without resorting to unnecessary mutual exclusion, thus providing fine-grained concurrency (ii) dynamic resource management. These constructs are supported by a *compositional proof system* for proving *interference freedom* of concurrently executing operations and to establish *live-ness* axioms and *deadlock-free* properties.

We state the following goals to drive our approach in specifying concurrency in UML.

a) To retain the classical UML constructs and semantics to the maximum extent



b) To provide UML extensions for expressiveness and precise operational semantics
c) To provide a higher level specification independent of execution environments
d) To provide fine-grained parallelism, synchronization, and mutual exclusion without using low-level synchronization primitives like semaphores and monitors
e) To support formal analysis of the system specification

3 INADEQUACIES OF CLASSICAL UML SEMANTICS TO SPECIFY FINE-GRAINED CONCURRENCY

In this section, we discuss inadequacies and inconsistencies in UML semantics through the UML-specification of the classical *readers-writers problem* with *writer* priority. As per the problem specification, a *writer* must have an exclusive access to the data while the *readers* can access the data concurrently.

UML objects can be *active* or *passive* as specified by the attribute *isActive* in metaclass ***class***. The meaning of *active/ passive* in UML is not entirely the same as in many object oriented concurrent languages [3]. In these languages, 'active' objects have internal control over concurrent requests made towards them where as 'passive' objects need external synchronization mechanism. In some of these languages for example; Eiffel, and ACT++, passive objects can only be used locally within an active object.

In UML, an *active* object has a thread of control and runs in its address space while passive objects run within the context of another active object which controls the caller [2]. In the absence of such a *controller*, passive objects execute their methods concurrently in the callers' thread of control and thus need explicit synchronization. In UML, though the semantics are not well defined, *passive* objects still have a degree of control over the invocations made towards them through the specification of meta-attribute *concurrency* with values *sequential, guarded,* or *concurrent* [2].

In general, *active* objects in UML follow *run-to-completion* semantics; no further messages are accepted until the current message is fully processed thus avoiding side effects due to interleaved execution of actions. The following issues can be identified from the UML specification of readers-writers problem. Readers, and writers are specified as active objects and we consider various options to specify the involved shared resource *Buffer.*

- *Buffer* as *active* object associated with a state machine: Parallelism can not be specified. For example, this specification precludes simultaneous reader activities due to inherent mutual exclusion of message processing resulting from *run-to-completion* semantics of the associated state machine. Thus the *Buffer* becomes internally sequential. Also deadlock issues arise when currently executing operation makes a synchronous operation call to it self. The only advantage of this choice is the implicit synchronization due to sequential execution.

- *Buffer* as *passive* object: The metaclass *Operation* has an attribute *concurrency* that specifies the semantics of concurrent calls to the same passive instance. The attribute can have one of the values: *sequential, guarded,* or *concurrent* dividing the operations of the class into three categories. By making operations *concurrent*, multiple invocations of an operation can exist simultaneously resulting in required parallelism. Though apparently this choice seems to satisfy the requirements of readers-writers problem, concurrency conflicts may arise requiring explicit synchronization mechanisms using low-level primitives like semaphores, monitors. Also passive object semantics is not clear in UML.
- In either of above choices, a well-defined event deferral mechanism is required. For example *write*, *read* events need to be preserved till appropriate state/ guard condition is reached. The availability of this mechanism is not clear particularly for passive objects as UML semantics of passive object and state machine conflict severely [4].

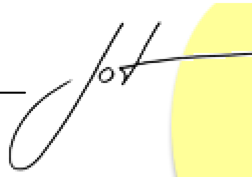
Other ambiguities and inconsistencies regarding concurrency in UML object model are well identified in [4, 5].

4 OVERVIEW OF CDL* CONSTRUCTS

Language constructs of CDL* provide modular specification with separation of concerns through MODSPEC, MODDES, and MODBODY parts of a module [1]. MODSPEC part provides unambiguous specification to both analyst and the implementor and facilitates modular verification. The syntax and semantics for the use of a module depend only on the MODSPEC part and is independent of other parts involving design (MODDES) and implementation descriptions (MODBODY). The constructs of the language provide high level specification of synchronization properties ensuring data integrity without resorting to unnecessary mutual exclusion due to low-level mechanisms like monitors. Also, the constructs are supported by the proof rules for the non-interference of concurrent procedures to establish the deadlock and live lock freedom of the programs.

The semantics of procedures and commands is described through designative phrases given in square brackets […]. From concurrency point of view, the main construct of a CDL* program is the *shared-resource* defining a set of shared data and a set of operations that can be performed by the processes on the data. The structure of the construct enables to synchronize processes, transmit data between them, and control the order of accesses by the processes to the shared data. We consider below the CDL* specification of readers-writers problem (as given in [1]).

ENTRY clause specifies the operations that are visible outside the module. PARALLEL clause specifies the procedures that could be instantiated concurrently. INVAR clause specifies the invariance of the resource as well as other properties e.g., liveness etc.



```
MODSPEC SHARED reader-writer

 EXPORT
      TYPE page : some_type;

 STATE
      VAR  buff: page;
      VAR  nr, nw: INTEGER INIT 0;
      VAR  writerbusy: BOOLEAN INIT FALSE;

 INVAR   [writerbusy → nr=0];

 PARALLEL (read, read), (startread, startread),
                 (endread, endread),
                 (read, startread, endread)
 TRANS

     ENTRY PROC startread
              [ nw=0 →  nr := nr+1 ];
     ENTRY PROC read(OUT x:page)
              [ x :=  buff];
     ENTRY PROC endread
              [ nr : = nr-1];
     ENTRY PROC startwrite
              [ nw : = nw+1;
               ¬ writerbusy AND nr=0 →
                      writerbusy := TRUE ];
     ENTRY PROC write(IN x:page)
              [ buff := x];
     ENTRY PROC endwrite
              [ nw := nw-1;
               writerbusy := FALSE];

END MODSPEC SHARED reader-writer;
```

```
MODBODY reader-writer
    PROC startread
              [ nw =0 →  nr := nr+1 ];
         DO DELAY (nw=0);
              nr := nr+1
       END startread;

    PROC read(OUT x:page)
              [x := buff];
         DO x:= buff;
       END read;

    PROC endread
              [nr : = nr – 1];
         DO nr := nr – 1;
       END endread

    PROC startwrite
              [ nw := nw+1;
               ¬ writerbusy AND nr=0 →
                      writerbusy := TRUE ];
         DO nw := nw + 1;
              DELAY ( writerbusy AND nr =0);
              Writerbusy := TRUE
       END startwrite;

    PROC write(IN x:page)
              [buff := x];
         DO buff := x
       END write;

    PROC endwrite
              [ nw := nw-1;
               writerbusy := FALSE];
         DO writerbusy := FALSE;
              nw := nw-1
       END endwrite;
END MODBODY reader-writer;
```

Fig 1. Specification of readers-writers problem under the assumption of trustworthiness of the processes

Specification given in fig.1 assumes that the processes are trustworthy. That is, the readers and writers follow the procedures in the order *startread, read, endread and sartwrite, write,* and *endwrite* respectively. The solution given is essentially the solution that can be described using monitors under the assumption of trustworthiness of the processes. It can be easily seen that if the processes do not call the procedures in the expected order then the program does not satisfy the specifications. Following access clause can be added to above specification making it valid without the assumption of trustworthiness of processes:

```
ACCESS   (startread)(read)(endread),  (startwrite)(write)(endwrite);
ACCESS   (startread)(read)*(endread),  (startwrite)(write)*(endwrite);
```

ACCESS clause defines the order in which the visible procedures could be accessed by the invoking processes. The order refers to the access order for each process and not just the general order of procedure invocation on the shared data. Access-expression is defined recursively through operators; nondeterministic choice (,), repetition (*), and

sequencing by concatenation.The second ACCESS clause above allows indefinite number of reads or writes after acquiring the resource. The access clause has wider perspective than that can be inferred from the above example, particularly for serial access devices (for complete discussion and more examples the reader is referred to the original work [1]).

5 PROPOSED APPROACH

Though UML provides constructs to model concurrency, many clarifications regarding semantics and semantic variation points are required for a consistent and unambiguous specification. Also the constructs and semantics related to concurrency are scattered in the official UML2.0 specification. In the proposed extension, we bring all the needed constructs and semantics within a well defined context, i.e. a shared resource. Where as many of the related works in UML [5, 10] are centered around refining the active object semantics, our work aims at refining the semantics of passive object to model a shared resource, a central entity for synchronization and communication. In our semantics, a shared resource is represented by an object which is externally passive and internally active. This choice presents high concurrency and protection of the integrity against concurrent calls. Also, this is essentially nothing but represents UML passive object semantics. The internal activism corresponds to refinement of UML semantics regarding meta-attribute *concurrency* which is not clear in UML. The proposed approach is based on the following extended UML meta-model fragment.

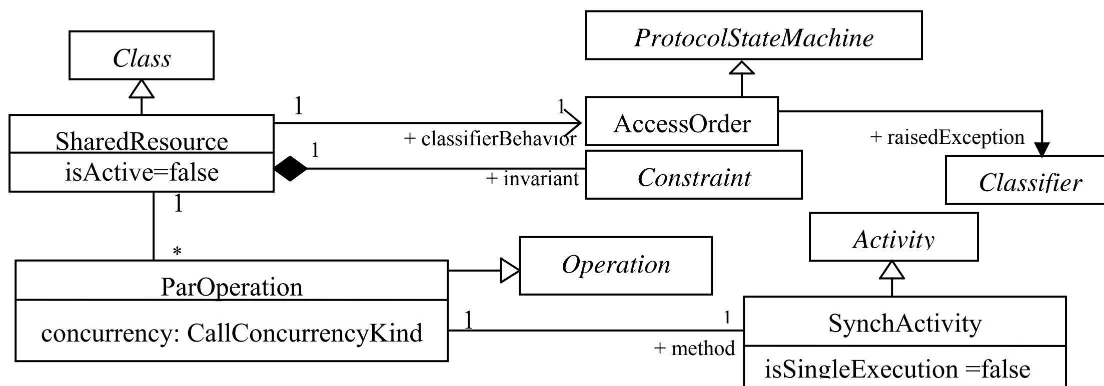
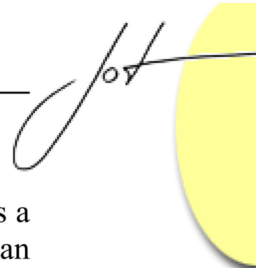


Fig. 2: An extended UML meta model fragment

SharedResource is the core element in the extended meta-model to specify a shared resource completely. It is a passive object (meta attribute *isActive* is set *false*). A *SharedResource* may contain invariants expressed as C*onstraints* which needs to be preserved at run time characterizing correct behavior of concurrent executions of the operations. Each operation of the *SharedResource* is of type *ParOperation* specifying



concurrent execution characteristics of associated methods. We further constrain that as a passive entity a *SharedResource* object can not respond to asynchronous calls and can only be associated with a protocol kind of statemachine defining the access order for invoking the associated operations. *ProtocolStateMachines* can be used to express the usage protocol of a part of the system. *AccessOrde*r is a *ProtocolStateMachine* associated with a *SharedResource* defining the order in which the visible procedures of a *SharedResource* could be accessed by the calling objects. We propose *run-to-completion* semantics for protocol transitions; the transition is only deemed completed once the associated method has fully executed. This solves many inconsistencies regarding UML semantics of passive objects and statemachines. The *AccessOrder* can raise an exception when the specified invocation order is violated by an invoking object and further behavior can be specified by the modeler.

AccessOrder is a very simple form of *protocol* statemachine with only operations associated with transitions and without complex features like concurrent regions, compound transitions etc. Here the protocol refers to the access order for each caller, and not just the order of operations that could be applied on the shared resource. In fact, it is this feature that distinguishes *AccessOrder* from that of usual protocol machines attached to classifiers. For example, if the *AccessOrder* specifies p1 and p2 in that order, then if an object, say A, has accessed the resource through p1(), then the next permitted operation by A on the resource is p2(). Furthermore, some other object, say B, will not be able to access p2() unless it has performed p1() even though A might has just finished the operation p1().

As the operation call events are the only external events received by a shared resource no event scheduling is required at object level. Hence, we preclude specifying guard conditions on transitions and an operation call always result in the execution of the associated method irrespective of whether the method is currently executing. This is again consistent with the UML's passive object semantics. This also resolves the semantics related to the event deferring mechanism in UML passive objects. As methods are *Behaviors* in UML, the necessary synchronization and mutual exclusion mechanisms can be specified within the method specification.

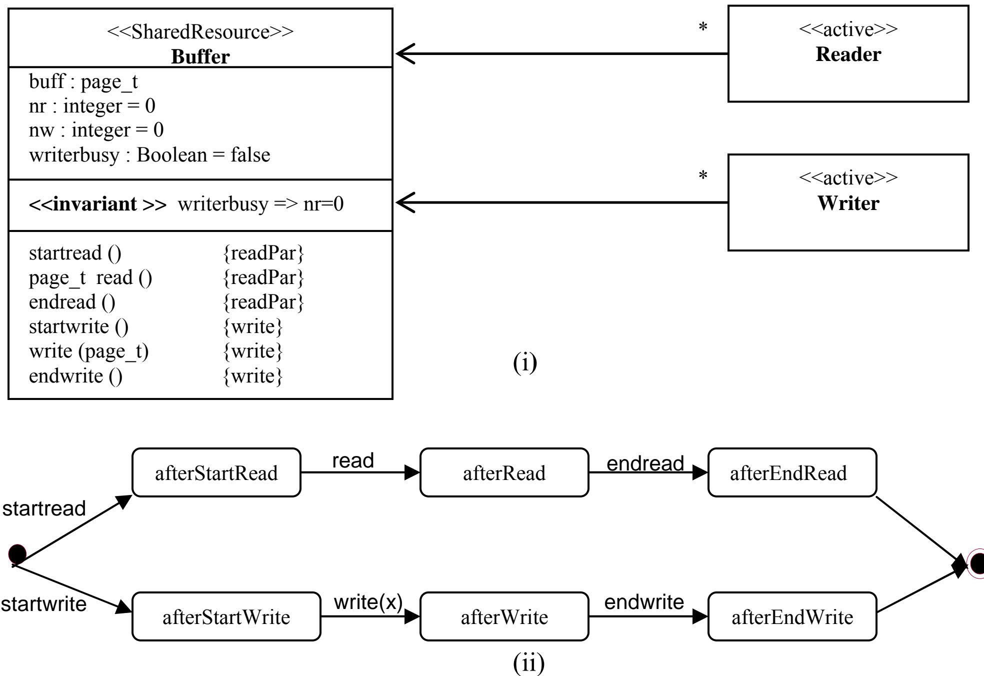
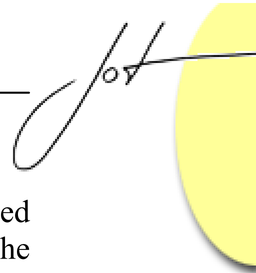


Fig. 3: Specification of analysis model for readers-writers problem
(i) Object diagram (ii) associated *AccessOrder*

The semantics of operation call mechanism for passive objects in UML is not well defined and is not sufficient for the *fine-grained* specification of plausibly concurrent operations. The metaclass *Operation* has an attribute *concurrency* with values *sequential, guarded, concurrent* to specify groups of operations which can be executed sequentially, atomically, or concurrently respectively. But the semantics of these mechanisms is not clear; sequentiality is not guaranteed by the model, integrity against concurrent calls is left to the modeler and *Guarded* operations semantics is not clear regarding the nature of blocking mechanism. Also the mechanisms are not sufficient for example, to specify whether the specification allows the multiple invocations of an operation. So we propose to extend *Operation* metaclass by ParOperation by redefining the associated *concurrency* attribute with new values of CallConcurrencyKind and associated semantics. This extension is very expressive to model various operation call mechanisms.

A *read* or *write* kind of operation specifies the multiple invocation nature of the operation where as 'par' indicates of the group operations that can execute concurrently. After invocation, the execution of *write* kind method is blocked until the previous execution is completed. We extend UML *activity* concept by *SynchActivity* to specify the complete behavior of methods of these operations.

Actions are the fundamental units of behavior in UML and are used to specify fine-grained behaviors. Their resolution and expressive power are comparable to the executable instructions in traditional programming languages. These actions are available



to higher-level formalisms like *Statemachines*, *Activities*, etc for describing detailed behaviors. As UML has not yet adopted a standard textual notation for actions (i.e. the standard action specification language, ASL), we use conventional programming language constructs to specify these actions in our extended *activity* diagrams (see fig. 5, 6).

Activity modeling in UML emphasizes the sequence and conditions for coordinating lower-level behaviors. The *activity* formalism in UML2.0 semantic framework provides advanced constructs for modeling complex control and data flow together with basic synchronization, and exception handling mechanisms. Even though the activities mechanism, with new Petri nets like semantics, is powerful enough to model complex activities it is still short of constructs to model various synchronization aspects. For example, there are no UML elements to model synchronization regions, mutual exclusion and conditional synchronization semantics.

We define *SynchNode* by extending *ActivityNode* in UML with a synchronization handler. These handlers can be referred by a unique identifier within the context of the containing shared resource. These handlers contain attributes *in* and *out* (synchronization counters) which are atomically updated whenever a thread of control enters or leaves the associated *ActivityNode*. A *SynchNode* can be associated with an optional entry/ exit condition, a boolean expression, which must be true for a control to enter/ exit the *SynchNode*. Thus a *SynchNode* can hold the control tokens till a condition is satisfied. Though this is in contrast with current UML 2 semantics where only data tokens can be held, such semantics for treating control as data, including queuing of control, is required and is called for in [12].

SynchNode represents a synchronization region within a plausibly concurrent method execution. A global invariant can be derived using *in, out* synchronization counters of all the *SynchNodes* of a shared resource. These invariants can be specified in the class diagram of the SharedResource (e.g. we assumed the implicit invariant pertaining to mutual exclusion of *read* and *write* kind of operations, which can also be specified explicitly using the synchronization counters). An approach to compose complex global invariant (GI) from existing GIs using GI patterns is discussed in [11]. This global invariant needs to be preserved for the interference freedom of concurrently executing methods. These *SynchHandlers* can be part of the execution record (a semantic varion point in UML2, [6]) of the corresponding methods or the SharedResource object itself.

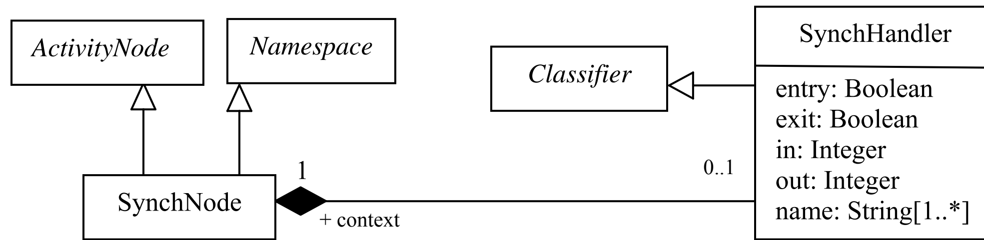


Fig. 4: An extended meta model fragment defining SynchNode and associated SynchHandler to model synchronization regions to specify conditional synchronization, mutual exclusion.

As a presentation notation, this handler can be attached with *ActivityNode* symbol as a dotted boundary containing optional name and optional condition(s). A group of atomic actions in an activity can be specified using dotted action node symbol. The counters *in*, and *out* of associated handler are implicit (see fig. 5, 6) and need not be specified.

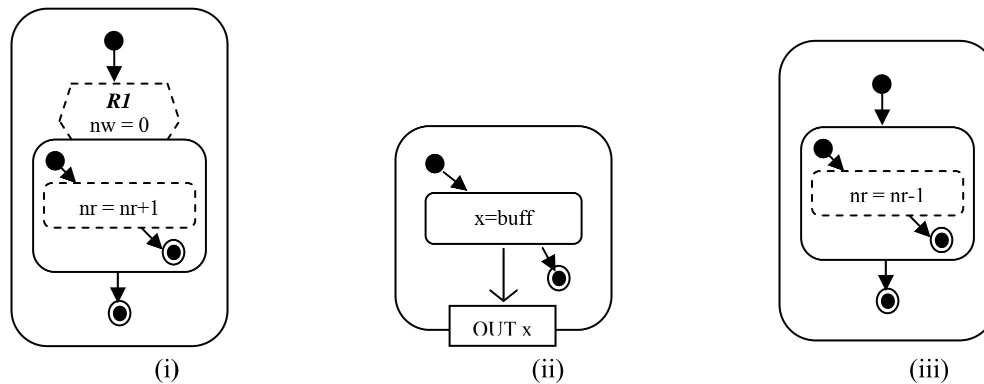


Fig 5: Extended UML 2 activity specification of *readPar* operations:
(i) startread() (ii) read() (iii) endread()

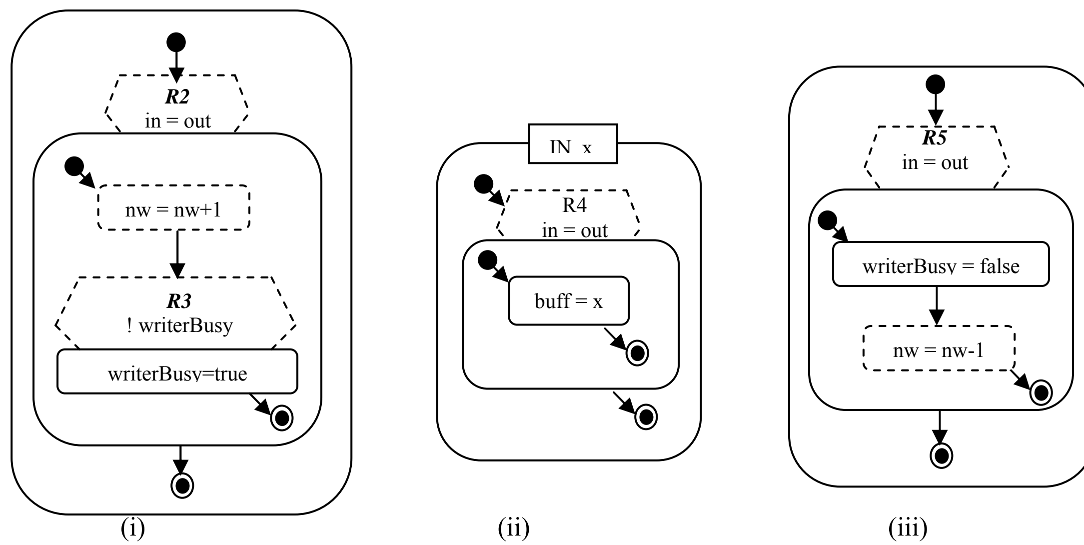
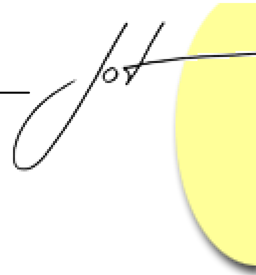


Fig. 6: Extended UML 2 activity specification of *write* operations:
i) startwrite() ii) write() iii) endwrite()



6 RELATED WORKS

To specify concurrency some UML methods and tools (RoseRT, Rhapsody, Accord) extend the semantics of UML active object model defining non-preemptive execution model (only one message processed at a time) or preemptive execution model (internal concurrency) with a *controller* for active objects. Also the internal concurrency of passive objects is controlled through protection mechanism like semaphores or through encapsulation inside an active object.

Shane Sendall and Alfred Strohmeier proposed an approach to specify concurrent operations through operation schema calculus based on OCL [7]. These schemata are declarative specifications of fine-grained concurrent operation behavior. Pre, post, invariant, shared resources, signals, and exceptions can be specified in operation schemas. The approach results in clear analysis model but the implementor has to obtain the necessary information from careful study of declarative OCL specification of operation schemas. The approach also uses protocol state machines to define temporal ordering of operations but constructing them requires complex state machine composition rules.

Charles Crichton et al. proposed a pattern for concurrency in UML [8]. The approach is based on modeling attribute states through state machine and operations states through activity diagram. The analysis model thus obtained can be converted to a formal process model (CSP) for validation of the design decisions using formal methods tools.

Sebastien Gerard et al. describe ACCORD/UML Methodology for modeling real time systems [9,10]. The approach defines a real-time object paradigm (RTO). In addition to attributes, and operations, a real time object consists of a mailbox, and a controller. The local controller is responsible for mailbox management, scheduling constraints handling, concurrency constraints handling, and thread management. The functionality is similar to that of an operating system scheduler. The behavior of an RTO can be described by a simple protocol kind of a state machine with no orthogonal composite states, no actions or activities in a state. Operations are classified as read, write, and parallel type and concurrent execution is allowed as per 1-writer/ N-readers protocol.

Iulian Ober proposed an approach to integrate an existing concurrent object model, named ATOM, with UML object model [5]. The proposed extension redefines active/passive object semantics to eliminate involved inconsistencies. Passive objects can not have statemachines. Active objects are quasi-concurrent: an executing method can explicitly yield the control for example while it is waiting for an event. Method invocation is de-linked from the associated statemachine and only signals are processed by the statemachine. The statemachine runs quasi-concurrently with the methods and is informed of methods start and end events.

Masaaki Mizuno et al. proposed a unified-process based aspect oriented methodology [11]. The approach defines steps to weave synchronization code into final solution. This is a semi-formal approach based on coarse-grained solution with formal

constructs and global invariants (GI). As finding suitable global invariants is not an easy task, GI patterns are defined to compose complex GIs.

7 CONCLUSION

UML 2 activities, with Petri net like semantics, is an expressive formalism to specify complex control/ data flow and hence suitable to model procedures and processes. In this paper, we have defined a UML-based approach to specify fine-grained concurrent access to shared resources without using low-level synchronization primitives and thus avoiding unnecessary mutual exclusion. This leads to better utilization of resources and overall system performance. We have cleanly integrated concurrency constructs with UML's active/ passive object model through a light weight extension of UML2.0 meta model. The proposed constructs of the approach are convenient for programming as well as for establishing the correctness of the specifications. We intend to extend the approach towards a comprehensive UML profile that can be easily mapped onto design and implementation models.

REFERENCES

[1] Shyamasundar R.K., and Thatcher J.W., "Language constructs for specifying concurrency in CDL* ", IEEE Trans. Software Eng. 15(8): 977-993 (1989)

[2] Object Management Group; "UML 2.0 Superstructure – Final Adopted Specification", OMG document, http://www.omg.org/docs/ad/03-08-02.pdf (2003)

[3] Papathomas. M., "Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming", Ph.D thesis, University of Geneva, 1992

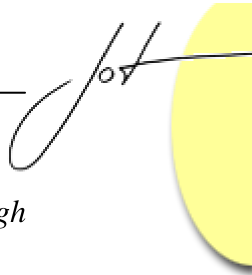
[4] Gerard. S., Ober. I., *Parallelism/ Concurrency Specification in UML*, white paper, UML Conference, Toronto, Canada, 2001

[5] Ober I., Stan I., *On the concurrent object model of UML. Proc. EUROPAR'99.*

[6] Selic, B., *On the Semantic Foundations of Standard UML 2.0*, Lecture Notes in Computer Science vol. 3185, Springer-Verlag, 2004.

[7] Sendall S., Strohmeier A., *Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML (UML 2001)*

[8] Crichton C., Davies J., and Cavarra A., *A Pattern for Concurrency in UML,* Oxford Computing Lab, (submitted in FASE 2002)



[9] Gerard, S.; Mraidha, C.; Terrier, F.; Baudry, B. *A UML-based concept for high concurrency: the real-time object*. In Proc of ISORC'04

[10] Gérard S., Nikos S. Voros, Koulamas C., Terrier F., Efficient System *Modeling for Complex Real-Time Industrial Networks using the ACCORD/UML Methodology*. International Workshop on Distributed and Parallel Embedded Systems (DIPES) 2000.

[11] Mizuno M., Singh G., and Nielsen M., *A structured approach to develop concurrent programs in UML.* In Proc. Third International Conference on UML, 2000.

[12] OMG Systems Engineering DSIG, UML for Systems Engineering RFP, http://www.omg.org/cgi-bin/doc?ad/03-03-41, March 2003

About the authors

Jagadish Suryadevara is currently a member of faculty, computer science group at Birla Institute of Technology (BITS), Pilani, Rajasthan, INDIA. His areas of research interests are in critical systems modeling and formal anlysis using UML. He can be reached at jagadish@bits-pilani.ac.in.

Shyamsundar R.K. is a senior professor and Dean, School of Technology and Computer Science at Tata Institute of Fundamental Research (TIFR), Mumbai, INDIA. He is a senior researcher whose areas of research interests include specification and verification of real-time distributed programs, semantics of concurrency, and logic programming. He can be reached at shyam@tcs.tifr.res.in.