View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Publications of the IAS Fellows

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2008**

Vol. 7, No. 8, Novmeber-December 2008

cmUML - A UML based Framework for Formal Specification of Concurrent, Reactive Systems

Jagadish Suryadevara, Birla Insitute of Technology and Science, INDIA
Lawrence Chung, University of Texas, Dallas, USA
Shyamasundar R.K., Tata Insitute of Fundamental Reseach, INDIA

Abstract

Complex software systems possess concurrent and reactive behaviors requiring precise specifications prior to development. Lamport's transition axiom method is a formal specification method which combines axiomatic and operational approaches. On the other hand Unified Modeling Language (UML), a *de facto* industry standard visual language, lacks suitable constructs and semantics regarding concurrency aspects. Though UML includes action semantics, its higher level constructs and object semantics are inconsistent. Motivated by Lamport's approach, this paper proposes a UML based specification framework '*cmUML'* ('cm' for concurrent modules) for formal specification of concurrent, reactive systems without object level diagrams and OCL. The framework integrates higher level diagrams of UML and addresses various concurrency issues including exception handling. It combines UML-RT and UML/ SPT profile as the latter defines a core package for concurrency and causality. Further the framework includes the characteristic safety and liveness aspects of concurrent systems. The proposed framework is in contrast with existing approaches based on low level primitives (semaphore, monitors). The paper includes several specification examples validating the proposed framework.

1 INTRODCUTION

In spite of proliferation of modern technologies, development of complex systems with concurrent, reactive behaviors remains a challenging task. The difficulty is largely due to the conceptual gap between the complex domains and the implementation technologies which requires formal yet intuitive specification languages and methodologies. There exists a pragmatic approach in formal specification of concurrent systems, for example Lamport's transition axiom method [Lam89, Lam00] which combines axiomatic and operational approaches for arriving at intuitive yet formal specification of complex systems. On the other hand, the visual specification languages like UML (Unified Modeling Language) and Model Driven Architectures (MDA) are emerging as new paradigm for development of complex

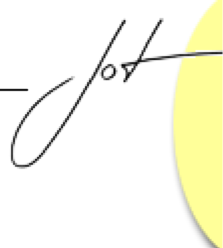
Jagadish Suryadevara, Lawrence Chung, Shyamasundar R.K: "cmUML – A UML based Framework for Formal Specification of Concurrent, Reactive Systems", in *Journal of Object Technology*, vol. 7, no. 8, November - December 2008, pp. 187 - 207 http://www.jot.fm/issues/issue_2008_11/article7/

systems. UML has become the de facto industry standard visual specification language [OMG02, Selic04]. Current UML methodologies for example COMET, CODARTS [Gomaa00], are largely based on informal design heuristics with focus on static aspects of the systems. Behavioral specifications of these approaches are largely representative lacking completeness and precise semantics. Further, these approaches specify concurrency using low level primitives like semaphores, monitors, and threads.

There exist formal approaches in UML with precise semantics, for example RT-UML [DJPV02], UML-RT [CG01], and UML/SDL [ITU00]. These approaches aim at real-time, embedded domains using a subset of UML features. For example, these approaches are largely based on statemachine semantics and do not integrate other behavioral aspects like data/ control flow, concurrency, synchronization etc. RT-UML provides semantical foundation to UML regarding concurrency, and communication. UML-RT and SDL/ UML are architectural approaches with focus on control-based reactive behaviors. But, none of these approaches handle higher level concurrency issues like multiple operation invocations, synchronization semantics. The proposed framework (named *cmUML* where 'cm' stands for concurrent modules) provides higher level architectural abstractions with precise operational semantics for specifying concurrent, reactive behaviors in terms of action, activity executions. As UML lacks formal semantics, *cmUML* provides much required unifying framework integrating action semantics, active/ passive objects, and higher level diagrams towards precise formal specifications (independent of design or implementation aspects).

To strengthen the emerging paradigm of visual specification languages and model driven architectures with the rigor of formal specification approaches, higher abstractions with precise semantics are required. *It is also necessary that such abstractions should be intuitive for the developers of the system*. Lamport's transition axiom method [Lam89, Lam00] (henceforth referred as TAM) recommends module based specification of systems where the modules (called 'components' in *cmUML*) are specified in terms of precise interface, and internal specifications. In this method, the internal specification resembles a higher level design of the module as the necessary system variables (PC, call stack, etc) can be used to represent the execution state. These variables are explicitly updated in response to module actions under safety and liveness constraints. The TAM approach is independent of any specification language. Its higher level design approach makes it convenient for adoption with UML framework. In this paper we extend UML (using its lightweight extension mechanisms) to define abstract architectural components with precise operational semantics. Further, these components are associated with UML's higher level diagrams retaining the benefits of its multi-view approach. In this regard, the main contributions of this paper are listed below.

1. Architectural abstractions for specification of concurrent, reactive, and flow behaviors under multi-view operational semantics (see appendix)
2. Component specification in terms of interface and internal specifications (representing an abstract implementation)



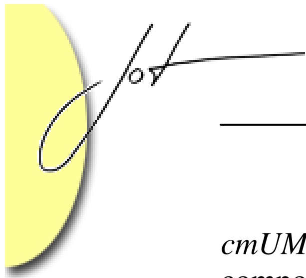
3. Semantical integration of higher level UML diagrams with the underlying object model
4. Specification constructs **GuardedAction**, **ServiceHandler** for expressiveness in concurrency, synchronization and exception handling semantics
5. Specifying component executions in terms of action and activity executions
6. Use case based **ScenarioContexts** representing interaction of internal behaviors with liveness semantics and explicit event ordering (forbidding race conditions)
7. Stepwise specification methodology for application of the framework

To implement TAM approach, the proposed framework combines UML-RT and SDL/ UML (for compositionality, formal semantics) and the UML/ SPT profile [OMG02] (for basic elements of concurrency and causality). The rest of the paper is organized as follows. In section 2, we give a brief overview of UML-RT, SDL/UML and SPT Profile. The proposed framework is defined in sections 3. Section 4 discusses a case study specification of vending machine, a classical problem in the literature. In section 5, we validate the profile through specification of classical concurrency patterns. We discuss related works in section 6. An approach for formal semantics definition is discussed in the appendix.

2 OVERVIEW OF UML-RT AND SPT PROFILES

The conventional UML mechanisms for specification of concurrency are: *active/ passive* objects, *concurrency* attribute of passive objects, *concurrent regions* of statechart, and concurrent actions. But as UML lacks a formal semantics, these mechanisms are not semantically integrated with the underlying object model resulting in inconsistent and ambiguous design models [GO01, Ober99]. UML-RT (similar to SDL/ UML in many aspects) is an architecture description language in UML. UML-RT (based on actor paradigm of ROOM language [SGW94]) defines architectural concepts as UML stereotypes. Specifically, it adds following stereotypes of standard UML elements (given in parenthesis) for modeling run-time structures.

- **Capsule** (Class): a basic building block that represents a complex *active* object with multiple interface points (ports) through which it interacts with its external environment. It contains sub capsules compositionally and associated with at most one statemachine (analogous to «system», «block» in SDL/UML).
- **Connector**(AssociationClass): a communication object that handles messages between ports attached to its ends (analogous to «channel», «gate» in SDL/UML).
- **Port**(Class): processes input or output of events
- **Protocol**(Collaboration): a specification of a closed group of participants (protocol roles) that interact in specific ways to accomplish tasks



cmUML adopts the notion of capsules, sub-capsules and ports of UML-RT (for compositionality) but not connector as it can be specified as an implicit association or as a component itself in case of complex associations (e.g. delaying channels).

The SPT profile (the standard UML profile for schedulabiloity, performance, and time [6]) was defined as a standard way to annotate the UML specification of real time systems towards automated quantitative analysis. It defines a generic resource modeling framework with abstract concepts allowing further extensions and mappings onto UML elements. Our decision to extend SPT is particularly relevant as the profile defines a concurrency package with abstract concepts like ***'ConcurrentUnit'***, ***'Scenario'***, ***'ActionExecution'*** etc. The core 'CoreResourceModelingFramework' package with the concurrency sub-package can be regarded as the kernel of the profile (see fig.1). Some important concepts of SPT profile as relevant here are: ***Instance*** and ***Descriptor*** (a run-time entity and its design time descriptor i.e. type); ***EventOccurence***, ***Scenario***, and ***ActionExecution*** define causality in the model; ***Scenario*** represents a sequence of actions (and sub-actions) with associated partial ordering representing concurrency.

cmUML extends the concepts of SPT profile with precise semantics and compositionality towards behavioral specification as an abstract architectural language (*a la* UML-RT).

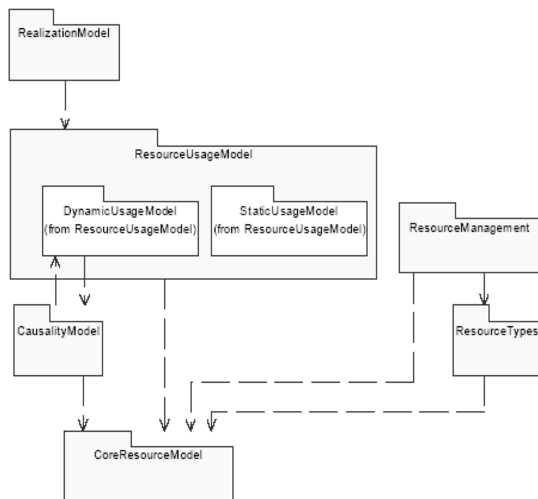


Figure 1.(a) GeneralResourceModeling package

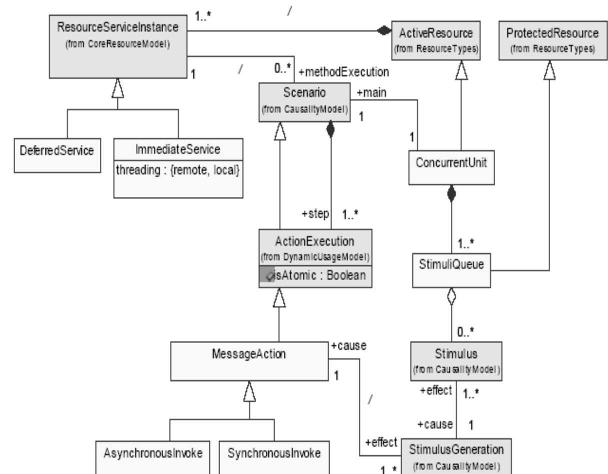
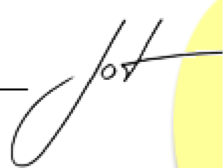


Figure 1. (b) Concurrency package

Figure 1. UML/ SPT profile packages at 'kernel' level

3 PROPOSED FRAMEWORK AND THE PROFILE

We follow SPT profile approach in defining the conceptual framework and the corresponding UML mapping. First we introduce conceptual elements of the framework in a class diagram notation (not related to UML metamodel) and then map



these onto UML metamodel using standard extension mechanisms. The conceptual diagram represents the basic abstractions and their relationships (see fig.2). We also describe the semantics informally (the formal description approach is outlined in the appendix).

In *cmUML*, a **component** is a generic entity (representing the **type** or **descriptor** of corresponding runtime instances) with specific functionality and behavior specified in terms of actions or activities under reactive or flow semantics. A component may be **concurrent** or **sequential** based on internal concurrency (i.e. concurrency is due to interleaved executions or alternating executions in run-to-completion). Depending on functionality and behavior the components are further classified as **system, state, port, service,** and **resource**. **System** component contains other components and responsible for their initialization. **Resource** component with abstract operations 'acquire', 'release', 'read' and 'write' represent a passive, protected data or hardware resource. **Resource** components with complex behavior may be specified as **system** components. **State** component represents reactive, synchronization, and exception handling aspects of internal executions. **Port** component represents interface specification with concurrency aspects, service access order, and inter component communication. **Services** are dynamic components instantiated in response to external requests in contrast to asynchronously executing **State**, and **Port** behaviors. An instance of a **service** may execute concurrently with itself and other compatible services. Action and activities are simple or **guarded** (with precise semantics). Guard expressions represent local **assertions** or global **invariants** representing synchronizations and exception handling in concurrent environment. A **ScenarioContext** represents interactions of component executions in response to external events with necessary liveness semantics and event ordering constraints.

In Table. 1 we define the corresponding UML profile using UML extension mechanisms stereotypes, tags and constraints (corresponding concepts of SPT profile are represented in *italics*). Also associations are represented via tags in *cmUML* as the profile does not use explicit associations. Stereotype or a UML name as tag type in the table indicates reference to the corresponding instance. Also absence of multiplicity indicates 0 or 1 where as * indicates 0 or more. The *cmUML* profile uses 'flat' versions of behavioral specifications i.e. activities, statecharts, and sequence diagrams without hierarchy as such features can be syntactically translated into equivalent flat versions. Also abstract methods are defined for a few abstractions (e.g. state, resource etc) to simplify the semantics description as well as make the specifications intuitive to system developers.

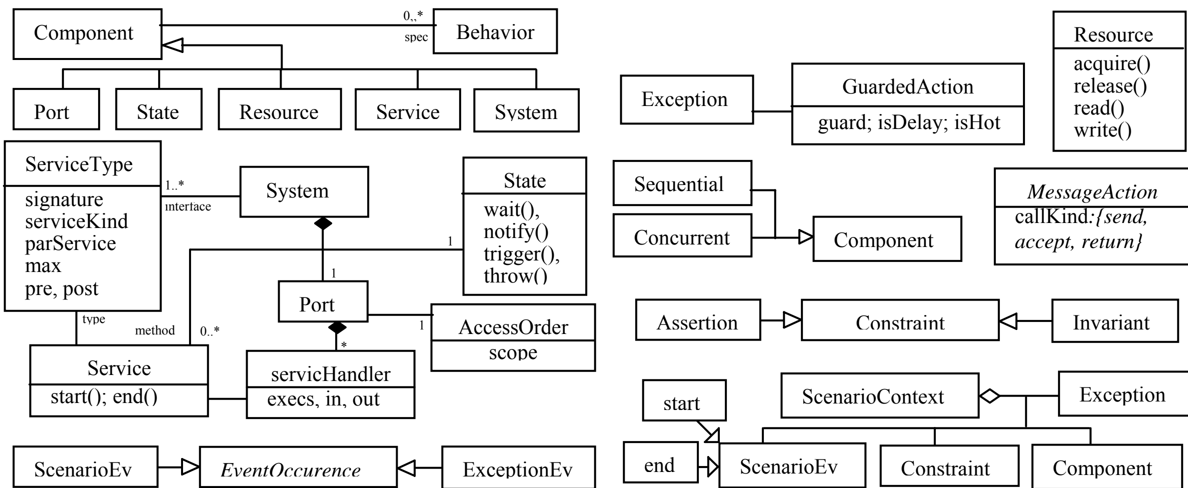
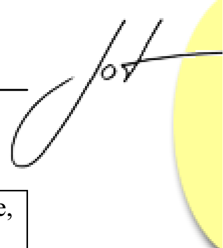


Figure 2. Conceptual model of the proposed specification framework

Stereotype (UML, SPT element)	**Tags[tagtype](multiplicity); Specialization/ Generalization; -- Constraints; (Description)**
Component (Class, *Descriptor*, *ConcurrentUnit*)	(Abstract) spec[Behavior](*); root[«system»]; concurrencyKind={*concurrent, sequential*}; evBuffer[«resource»] **Specializations** : system, port, state, service, resource
«system»	port[«port»]; state[«state»]; service[«service»](*); -- port, state are not *null*
«port»	interface[«serviceType»](*); spec[«AcessOrder»]; handles[«serviceHandler»](*); policy={*FIFO, Priority*} -- concurrencyKind= 'sequential'; port, state are *null*
«state»	spec [«Reactive»]; -- concurrencyKind= 'sequential'; port, state are *null*
«service» *(Scenario)*	spec [«Flow»]; (dynamic component of the system) -- concurrencyKind= 'sequential'; port, state refer to those of its root --evBuffer is *null* **Generalization**: ActivityExecution
«resource» *(ProtectedResource)*	(protected entity that need to be (atomically) acquired and released for read or write kind of accesses. Resources with complex behavior can be specified as «system» components)
ServiceType or ST (Operation)	max[integer]; serviceKind={*read, write*}; parService[«serviceType»](*); params[string]; (where string can be interpreted using BNF like grammar)
ServiceHandler or SH (Classifier)	execs[«service»](*); in(integer); out(integer); -- service executions corresponding to a service handler are of same serviceType
GuardedAction or GA (Action)	guard[bool]; isDelay[bool]; isHot[bool]; exception[«exception»];isAtomic[bool] **Generalization**: GuardedActivity
ActivityExecution or AE (*Action Execution*)	(Abstract) ('Activity' is a sequence of atomic actions with possibly partial ordering)
Exception (*Stimulus*)	(service execution containing exception raising action terminates)
MessageAction or	synchKind={*send, accept, return*}



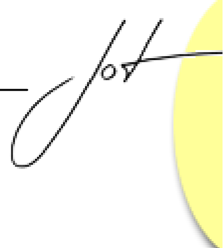
MA	(corresponds to asynchronous call, synchronous call till message acceptance, syncrhonous till result returned or service completed)
ScenarioEv (*EventOccurence*)	eventKind={ *start, end*} (these events represent start and end of a service execution)
Exception (*Stimulus*)	(service execution containing exception raising action terminates)
AccessOrder or AO (BehaviorStateMachine)	scope: {*local, global*} ('scope' specifies whether the access order is applicable globally or per client)
Reactive (BehaviorStateMachine)	Represents the reactive behavior of «system» component asynchronously executing with «service» components
Flow (Activity)	Represents the data and control flow behavior of «service» components
ScenarioContext or SC (Sequence)	(specification of behavior service interactions in response to external requests with liveness semantics)
Assertion, Invariant (Constraint)	(Assertion –a constraint over local data Invariant –a constraint over global data e.g. incarnation counters of service handlers)

Table 1. cmUML profile for the proposed framework

cmUML Semantics Description

In the rest of this section, we informally describe the semantics of the cmUML specificaitons (see appendix for formal description approach). One of the main constructs defined in the profile is **ActivityExecution** as a generalization of SPT Profile's ActionExecution (consistent with UML definition of activity as an action). Activities are at a higher granularity than actions and represent a service. A service is associated with a run-time handler 'ServiceHandler' in 'port' component with information regarding service instances that started and completed (using incarnation counters ***in*** and ***out***). This information can be used to specify complex synchronization patterns in the form of global invariants representing safety conditions in a simpler way [JS07, Miz99]. A set of useful global invariants are proposed [Miz99] which work as basic patterns to compose appropriate global invariants for specifications. Translations exist from global invariant based coarse-grained specifications to fine-grained synchronization code using semaphore, monitors etc. Another important construct defined in the profile with respect to concurrent execution characteristics is **GuardedAction**. This allows specifying precise semantics corresponding to the guard evaluation and the execution of the corresponding action or activity (see fig.5). The GuardedAction specifies synchronization (i.e. wait semantics) and exception handling. The exceptions are handled by corresponding 'state' component or thrown into higher level 'state' components (*a la* java *try-catch* block). Thus GuardedAction provides much needed specification construct for synchronization, exception handling behavior of sequential executions in concurrent environment [Lohr92]. Communication aspects of *cmUML* components are externally message based (suitable for distributed environment) and internally message, or shared resource based.

- **System**: the main abstraction which contains other components compositionally and associated with its initialization behavior. It has sub components of type 'port', 'state' and 'service'. The 'port' component represents its interface specification and 'state', 'service' components represent its internal specification (*corresponding to an abstract implementation and a higher design specification of the component*). The 'port' and 'state' are static components where as 'service' components are dynamic corresponding to external requests. A 'system' component may also contain 'resource' type components to specify protected, shared resources.
- **Resource**: represents a 'simple' protected shared resource with methods 'acquire()', 'release()', 'read()', and 'write()'. A resource instance is explicitly 'acquired' and 'released' (atomically). Resources with complex internal behaviors can be specified as 'system' type components.
- **Service**: the dynamic behavior corresponding to an interface 'ServiceType' of a component invoked through associated port, specified with data and control 'flow' semantics (an activity diagram). The concurrent nature of a ServiceType with itself and other compatible 'ServiceTypes' is specified by tags 'ServiceType', 'serviceKind'. Events 'start' and 'end' are generated corresponding to a service execution (event 'end' not generated if the service is terminated due to a raised exception). These events are broadcasted to all state components with in the scope of the containing top most 'system' component.
- **Port**: the interface specification of concurrent and reactive behaviors of a component as observed externally. As recommended in Lamport's approach, the interface can be specified with precise operational semantics. It exports a collection of 'ServiceTypes' with concurrency annotations through associated tag values for specifying concurrent semantics of invocations. It enforces 'pre' conditions, if any, for 'ServiceTypes' where as internal specification gurantees the 'post' conditions. It also handles inter-component communication aspects. The associated 'AccessOrder' behavior (a behavior statemachine) specifies the invocation order of the services (i.e. *temporal ordering dependencies* among the specified services) as well as the abstract statespace of the component. The 'AccessOrder' is an important abstraction addressing many issues of concurrent systems [JS07]. For a concurrent component, this also aids in identifying sub components (see next section).
- **State**: specifies the reactive, coordination, exception handling aspects of internal behaviors of 'system' component. The associated '*Reactive*' behavior (specified using behavior statemachines) executes asynchronously with respect to its services. Thus a 'system' component associated with a 'state' behavior represents an abstract monitor with concurrent threads of control (classical monitors cause unnecessary mutual exclusion [JS07]). Though it corresponds to 'AccessOrder' specification of the corresponding 'port' component (i.e interface specification) it may contain additional abstract states, transitions, and activities (*a la* Lamport's *stuttered* transitions corresponding to an



implementation specification). Methods *wait* and *notify* facilitate service synchronization (*a la* classical monitors). Further a 'state' component receives events '*start*', '*end*' corresponding to service executions.

- **ScenarioContext**: corresponding to each use case, 'ScenarioContexts' represent interaction of internal behaviors with liveness semantics inspired from LSCs (Live Sequence Charts) [DH99]. These contexts specify message, event exchange, and coordination in response to external stimuli (events or invocations). Sequence charts with liveness semantics support the verification of component properties [ITCB04]. In *cmUML*, these contexts essentially represent the principle behaviors of the system without error scenarios (corresponding to failure of pre conditions or guard expressions) and latter can be 'plugged-in' to specificaitons through exception handling mechanism where activities corresponding to exceptions are invoked by the corresponding 'state' component (or 'thrown' into higher level 'state' components).

4 A SPECIFICATION METHODOLOGY

In this section, we propose a step-wise specification methodology for the application of *cmUML*. The methodology assumes use case based requirement analysis and a higher level decomposition strategy for arriving at the initial subsystems [Goma00]. For the case study below there is only one subsystem which can be taken as the initial «system» component. For a complex system there may exist many subsystems for which the methodology can be applied separately. We describe the specification approach in terms of the following tasks.

Task1: Identify the component interface with services offered. The information can be obtained from requirement artifacts like problem statement, use cases, and context diagrams.

Task2: Determine the concurrent execution behavior of interface services (*serviceKind* and other tags) and their temporal ordering dependencies as observed externally. This information is specified as the 'AccessOrder' behavior of the corresponding 'port' component. AccessOrder is a behavior statemachine and transition guards may include expressions over incarnation counters of 'ServiceHandlers' corresponding to interface 'ServiceTypes' of the 'system' component. The AccessOrder specification also aids in component decomposition as explained next.

Task3: Considering the information obtained in above task, perform the component (or subsystem) decomposition to find the internal (behavioral) structure by dividing the interface services into a set of concurrent groups of services. This decomposition can be fine-tuned by applying the general task cohesion principles from OOAD approaches (e.g. functional cohesion) [Goma00]. Each of these concurrent groups can be specified as a subcomponent. For simple components with no internal structure, this step is skipped.

Task4: Corresponding to each use case, specify one or more 'ScenarioContext' involving interaction between 'system', 'service', 'port', 'state', and 'service'

components with liveness semantics and explicit event orderings. Also, the control and reactive aspects are specified as a 'state' component using behavioral statemachine and may be represented in ScenarioContexts of the component.

Task5: Specify services of 'system' components as UML activity diagrams (flows), identifying the functions to be implemented and specifying 'Guarded' actions or activities (and associated atomicity) if any.

Task6: Further refine the 'service', 'state', 'ScenarioContext' specifications by identifying synchronization, exception handling aspects among the concurrently executing 'services' and 'state' components. This includes identifying appropriate invariants (by identifying 'guarded' actions or activities), and exception handling activities for the 'state' component.. This task also includes identification of external/ internal events and component responses.

Task7: Repeat above tasks for 'system' sub components identified in task2.

We elaborate above tasks with a case study. Consider the UML specification of a vending machine, a well known specification example in the literature for example in [ITCB04]. A vending machine (VM) accepts coins from users to dispense a drink of chosen choice. The user gives coins, one at a time, and when the sum is sufficient enough the corresponding choices of available drinks are displayed. The user can select any of enabled choices. The drink and the extra coins, if any, are dispensed (for simplicity, we assume that the VM doesn't remember the coins of previous transactions). Also the user's request to cancel the transaction may be considered.

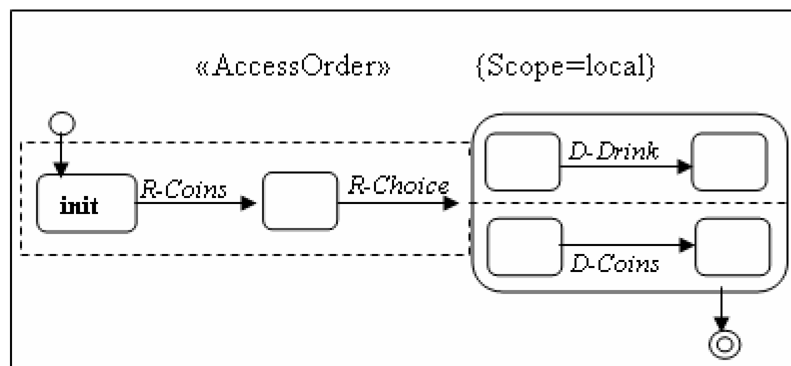
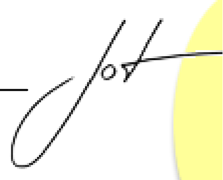


Figure 3. «port» specification of temporal dependencies among the interface service invocations for VM

Task1: Interactions of the system with its environment (i.e. user) is considered. The user '*gives*' sufficiently more coins and when prompted by the VM '*selects*' his choice of the drink. The VM, after '*validating*' the choice and the received coins, '*dispense*s' the 'drink' as well as the 'balance' coins if any. From the first analysis of external interaction we can observe four main services of the VM, involving its environment (user): *ReceiveCoins*, *ReceiveChoice*, *DispenseDrink*, and *DispenseCoins* (denoted concisely as *R-Coins, R-Choice, D-Drink, D-Coins*) (see fig. 3).



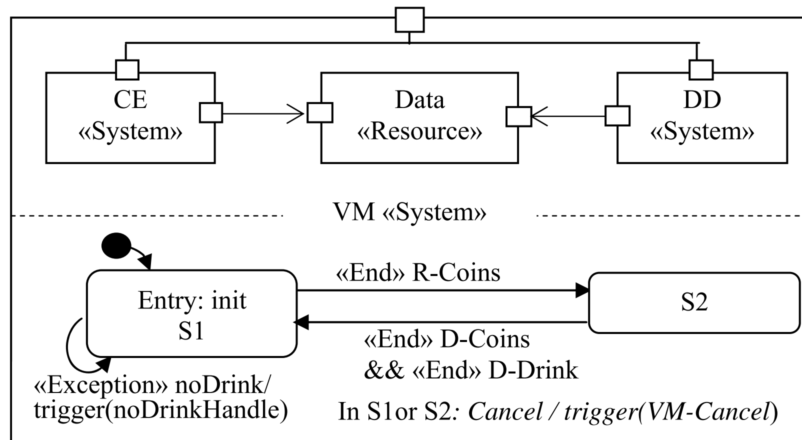


Figure 4. Structure and «state» specification of «system» Component VM

Task2: During this task, we determine the possible concurrency and temporal dependency between the component services as observed externally. This is specified as 'AccessOrder' in figure 3. The specified tag value 'scope' is redundant in this example as only a single user is involved at a given time. The complete interface specification («port») includes concurrency aspects of all ServiceTypes of the component (as observed externally). For VM, all service types have similar tag values {isAtomic=*false*; serviceKind=*write*; max=1} with additional information that *D-Drink, D-Coins* may execute in parallel. The semantics of these tags is also operationally specified in the corresponding 'AccessOrder' specification.

Task3: From figure3 we observe a concurrent region and a sequential region in dashed border. Now following task cohesion principles of OOAD approaches [Goma00], we can identify two concurrent components with functionally related services; a 'Cash Exchanger' component (CE) (that handle interface services *R-Coins, D-Coins*) and a 'Drink Dispenser' component (DD) (that handle interface services *R-Choice, D-Drink*). The temporal dependencies specified as part of interface specification need to be preserved in the internal specification. This is done through the specification of «state» component of VM in figure 4.

Task4: For simplicity we assume 'Env' represents the 'port' component and includes the hardware interfaces through which user interacts with VM (e.g. choicePanel, coinSlot, etc). Now, we can specify the ScenarioContexts of VM (see fig.6 which includes all the contexts for brevity). Only specified events under given liveness constraints are of interest to the context with respect to system behaviors which may include other 'unspecified' internal events, actions etc (*a la* Lamport's 'stuttered' transitions). In this specification, all solid notations, for example life-lines or segments there of (representing executions) and message actions, represent compulsory or liveness notion of mandatory behavior while dashed ones represent optional behaviors [14].

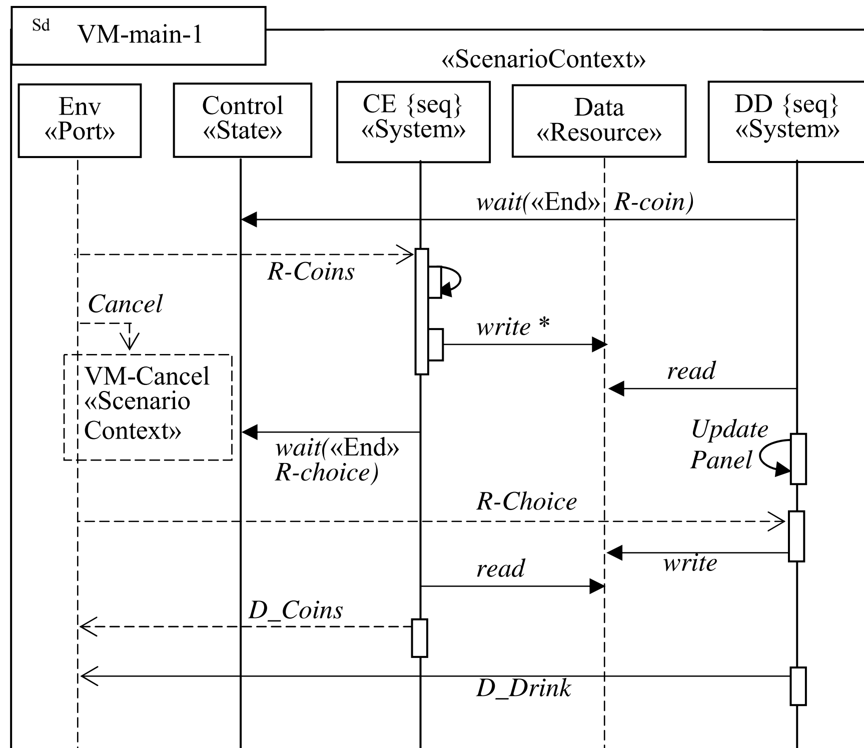
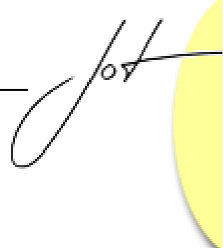


Figure 6. A «scenarioContext» specifies primary behavior of VM component with liveness semantics

Task5: The computational aspects of component services are specified in terms of the implementation level activities using activity diagram (with data and control flow semantics). Fig.5 specifies the 'flow' behavior of the 'service' R-Coins with necessary guarded semantics. The associated tag values specify that the service execution does not wait for guard value to become true and terminate by raising an exception. 'GuardedActions' are useful to specify atomic update of shared data values (isAtomic=*true*) or synchronization semantics regarding guard evalution. In this context atomicity indicates that the guard value can not change during execution of the action(s) (in fig.5 the outer guard corresponding to drinks availability cannot change during execution of R-Coins behavior). Also a guard expression may declaratively specify a condition referring to old and new values of shared data using notation e.g. x@preAU and x@postAU enhancing expressiveness of specifications [Lam00].

Task6: Now we can further refine the various specifications. We can identify '*guardedActions'* with guards as invariants over incarnation counters *in, out* of serviceHandlers of component ServiceTypes specifying service synchronizations if any (This is required while compiling higher level specifications with internal activities that need to be synchronized. These activities can be specified as separate 'internal' services). We can also specify exception handling activities by extending «state» specification. We can also identify external, and internal events and specify corresponding event handling activities. In fig.6 we identified possible external event '*Cancel'* due to user's interaction with VM. A new ScenarioContext VM-Cancel is



specified. Multiple contexts may be 'enforced' in parallel i.e. in 'interleaved' fashion, over system behaviors (this is possible as 'system' components do not discard events implicitly).

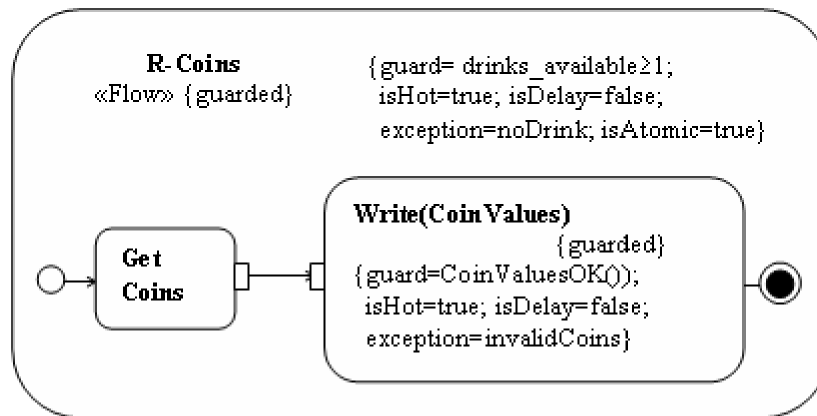


Figure 5. Activity specification of a service with *guarded* semantics

Task7: We can complete the specification of VM by specifying each «system» sub component (i.e. CE, DD) following previous tasks. We skip these steps here.

5 FRAMEWORK VALIDATION: SPECIFICATION OF CLASSICAL CONCURRENCY PATTERNS

In this section we specify classical concurrency patterns to show merits of *cmUML* approach over current UML approaches (using low level constructs for e.g. [GE04]). These approaches use low-level primitives like locks, semaphore, monitors etc to describe concurrent behavior where the semantics of these constructs are either not specified or specified in complicated OCL statements. Also in these approaches, though higher level diagrams are used, no precise semantics can be inferred about behaviors of the system specified. In contrast, *cmUML* specifications are precise without using low level primitives.

Readers-Writers Synchronization Pattern

In fig.7 we have shown the specification of the pattern in current UML practices (without OCL statements) and in fig.8 using *cmUML* approach. The proposed approach retains the abstractness of specifications yet providing precise operational behavior. The specification in fig.8 does not use many proposed abstractions (i.e. state, service, scenarioContext) as the pattern is simple and services are primitive. Also the pattern does not possess any reactive or state-based behavior. Hence the interface specification (i.e. 'Port', 'AcessOrder') it self is sufficient. From the specification it is precise that multiple readers are allowed where as a single writer executes in mutual exclusion. Also by 'policy' tag value i.e. as '*FIFO*' with «Port», there is no starvation of 'writers'.

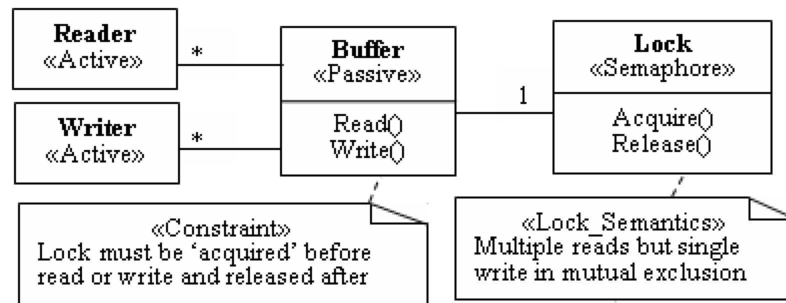


Figure 7. Specification of Readers-Writers synchronization pattern in UML approaches

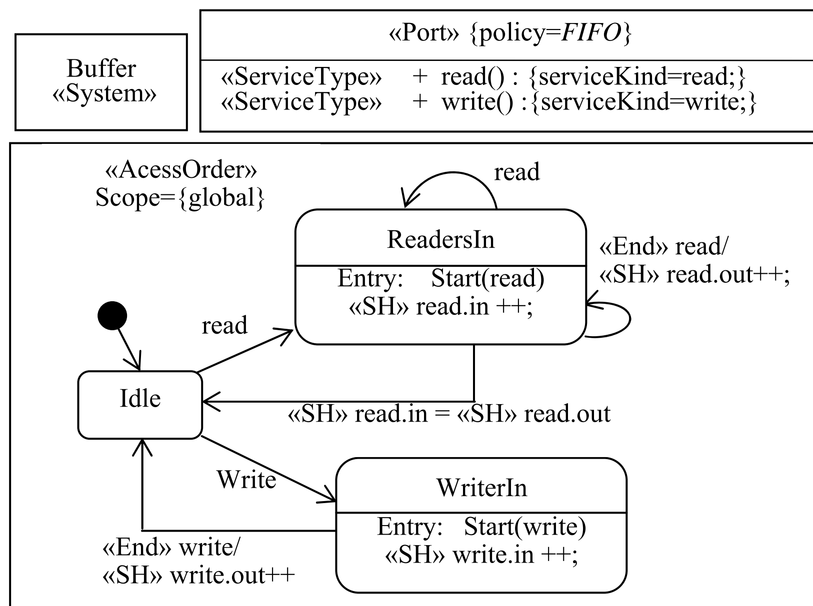


Figure 8. Specification of Readers-Writers synchronization pattern in *cmUML*

For 'ServiceTypes' parameters can be specified using tag 'param'(a string defined in BNF form: {**in** | **out**: variableName(variableType)}).

Producer–Consumer Synchronization Pattern

The pattern has state based behaviors. Fig.9 specifies the pattern (invocation behavior of get() vomited) in current UML approaches where as fig.10 is the corresponding *cmUML* specification. Though fig.9 includes behavior specification using UML sequence diagram, it is not formally precise.

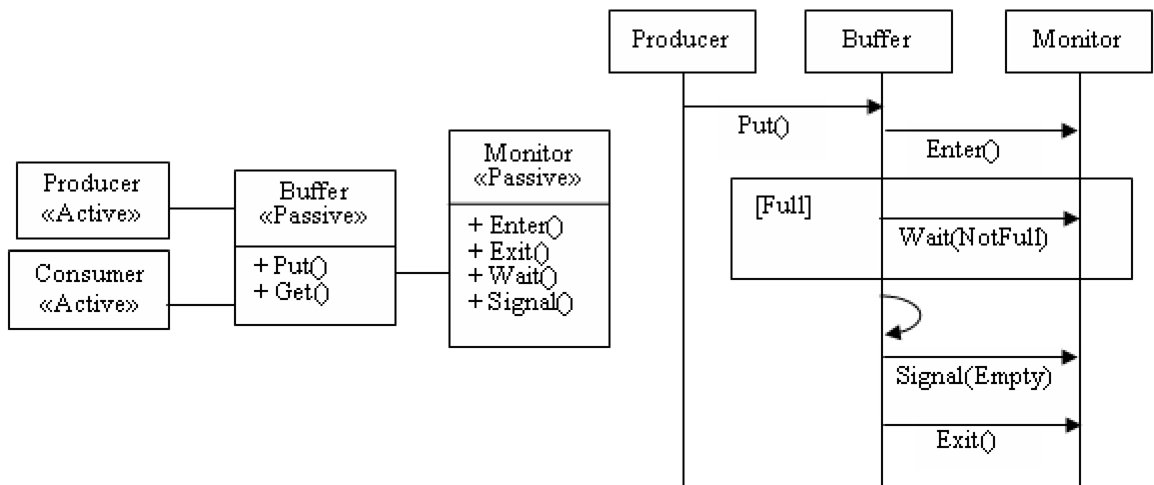


Figure 9. Specification of Producer-Consumer synchronization in UML approaches

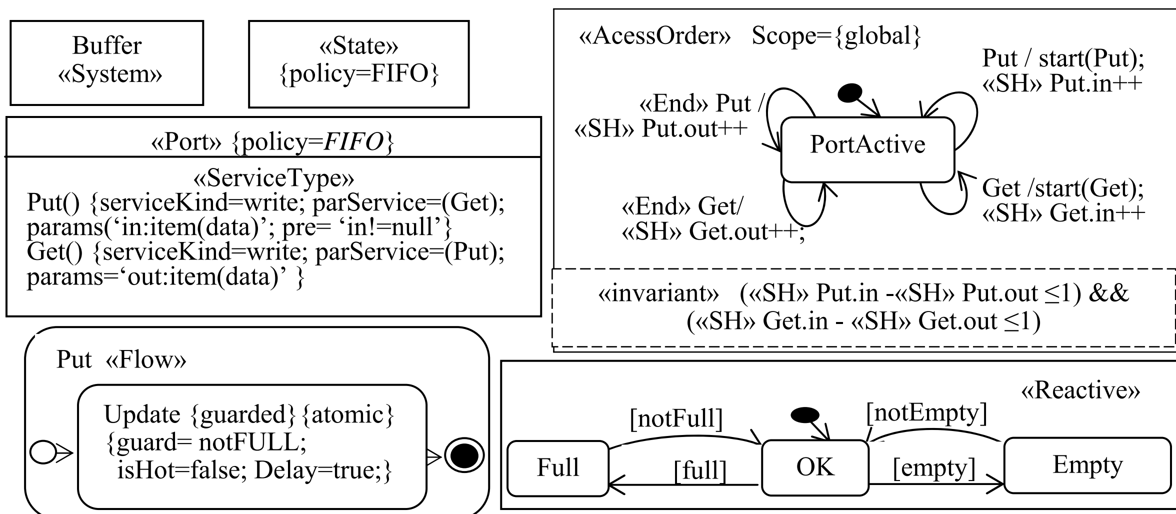


Figure 10. Specification of Producer-Consumer synchronization pattern in cmUML

The *cmUML* specification of the pattern (fig.10) uses many abstractions of the proposed framework. «Port» and «AcessOrder» specify invocation behavior of interface services. «Reactive» part specifies the state based behavior. «Flow» specifies the important part of the specification i.e. sequential behavior of the «service» Put. It contains the *guarded activity* 'update' under guard '*notFull'* with the specified atomicity indicating that the guard value can not change during execution of the activity. Also the tag values isHot=false and isDelay=true specify that the service execution waits till the guard is satisfied (possibly forever!). The new specification does not use ScenarioContexts as they are not required in this example.

6 RELATED WORKS

There exist several works involving concurrency in UML. We have already discussed UML-RT and UML/SPT Profile. In this section, we briefly look at other works [Ober99, CDC02, DM99, Omar03, DJPV02]. None of these works comprehensively address the issues of concurrency specification from formal specification perspective.

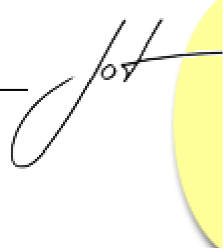
Charles Chrichton et.al. proposed a pattern for concurrency specification in UML [CDC02]. This pattern addresses specification of multiple instantiations of operations on an object. The specification approach separates (non-atomic) operation specifications from statemodel specification of an object using different diagrams (i.e. activity and statmachine). This is a formal approach where the operation, and statemodel specifications are converted to CSP process specifications and effects of concurrent executions of operations on the object are examined using FDR model checking tool. But, the approaches based on translation into formalisms fall short of covering the rich range of features in UML.

Iulian Ober and Ileana Stan proposed a quasi-concurrent object model for UML's active objects by integrating an existing concurrent object model, namely ATOM, with UML [Ober99]. The proposed extension redefines active/ passive 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 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 notified of method start and end events. Some aspects of the approach are related to *cmUML*.

A UML package for specifying Real-Time objects was proposed [DM99]. The constructs of the approach are based on the objects of the RTSORAC (Real-Time Semantic Objects Relationships and Constraints) model. Concurrency in an object is determined by the Compatibility function (represented as a matrix). Each function parameter is specified as 'read' or 'write' type and compatible functions (i.e. those functions which can execute concurrently) are determined based on their parameter values. The approach causes more overhead but can increase the potential parallelism and thus may be helpful on certain parallel architectures. The aspects of the compatibility function can be found in *cmUML*.

Aspect oriented approaches are being proposed for modeling as well. For example, Omar Aldawaud et.al. proposed a UML profile for aspect oriented software development in UML [Omar03]. The profile defines stereotypes «aspect», «crosscut» etc. Various execution aspects of an object, for example synchronization and exception handling, can be specified as separate «aspect» objects which can 'crosscut' into functionality of the main object in synchronous or asynchronous manner.

Werner Damm et.al. defined a subset of UML, *krtUML* [DJPV02], which is rich enough to express all behavioral modeling entities of UML for real-time systems with formal interleaving semantics using symbolic transition systems (STS) providing the



much needed semantical foundation for formal verification of real-time UML models. This work addresses wide range of language issues of UML as well as critical issues of concurrency, and communication. While *krtUML* is concerened with the complete specification of systems at lower granularity (with concurrency due to concurrently executing sequential objects), *cmUML* addresses similar issues at higher granularity of specifications (i.e. abstract architectural components) with intra component concurrency.

7 CONCLUSION AND FUTURE WORK

In this paper we have proposed *cmUML*, a behavior specification framework in UML, by combining and extending the concepts of UML-RT and UML/SPT profile. UML-RT is an architectural description language with focus on real-time embedded systems. The SPT profile defines abstract concepts to annotate models for real time systems towards quantitative analysis. *cmUML* adds compositionality, and precise semantics over SPT profile to define a behavior specification language. The framework covers wide range of concurrency issues including exception handling. It further adds the safety, liveness aspects of concurrent computations. It also retains the multi-view approach of UML by integrating behavioral diagrams (statecharts, activity, and sequence diagrams) in underlying semantic framework. The proposed framework is independent of class diagrams and OCL (UML's object constraint language). Also *cmUML* is independent of design aspects like active, passive objects to specify concurrency.

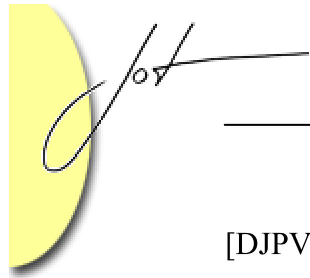
The proposed framework is motivated by Lamport's transition axiom method for formal specification of concurrent systems and shows the formal rigor of the method can be followed with UML. As *cmUML* is based on formal semantics (see appendix and [JS06]) the formal verification techniques can be supported [WMC01, ITCB04]. We intend to further refine the elements of *cmUML* to make it a formal specification language for Lamport's transition axiom method to obtain the benefits of Lamport's verification techniques in industry standard specificaiton enviroments like UML.

REFERENCES

[CDC02] Crichton C., Davies J., and Cavarra A., "A Pattern for Concurrency in UML", Oxford Computing Lab, submitted in FASE 2002

[CG01] Shang-Wen Cheng, and David Garlan, "Mapping Architectural Concepts to UML-RT", *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, Las Vegas, USA, June, 2001.

[DH99] Werner Damm, David Harel, "LSCs: Breathing life in to Message Sequence Charts", In Porc. 3rd IFIP International Conference on Formal Methods for Open Object-based Distributed System, 1999



[DJPV02] W. Damm, B. Josko, A. Pnueli, and A. Votintseva., "Understanding UML: A formal semantics of concurrency and communication in real-time UML", In Proceedings of FMCO'02, LNCS. Springer Verlag, November 2002

[DM99] L. DiPippo and L. Ma, "A UML Package for Specifying Real-Time Objects", University of Rhode Island, Technical Report, TR99-274, Nov. 1999

[GE04] A. Goni, Y.Eterovic, "Building Precise UML Constructs to Model Concurrency Using OCL", Proc. UML 2004 Conference, LNCS Vol 3273, pp 212-225, 2004

[GO01] Gerard. S., Ober. I., "Parallelism/ Concurrency Specification in UML", white paper, UML Conference, Toronto, Canada, 2001

[Goma00] H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison-Wesley, USA 2000

[ITCB04] Ingo Schinz, Tobe Toben, Christian Mrugalla, Bernd Westphal, "The Rhapsody UML Verification Environment", Proc. of 2nd Int. Conf. on Software Engineering and Formal Methods (SEFM'04), Beijing, China, pp 174-183, 2004

[ITU00] ITU-T, "SDL combined with UML", ITU-T recommendation Z.109, 2000

[JS06] Jagadish. S., Shayamasundar R.K: "cmUML- A Precise UML for Abstract Specification of Concurrent Components", Proceedings of 18th International Conference on Parallel and Distributed Computing and Systems (PDCS), Dallas, USA, Acta press, November 2006, pp 141-146.

[JS07] Jagadish. S., Shyamasunder R.K: "An UML-based approach to Specify Secured, Fine-grained, Concurrent Access to Shared Resources" Journal of Object Technology (JOT), vol.6 no.1, Jan-Feb, 2007, pp 107-119. http://www.jot.fm/issues/issue_2007_01/article3/

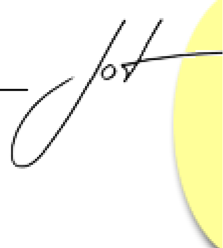
[Lam00] Leslie Lamport, "A Formal Basis for the Specification of Concurrent. Systems", Notes for the NATO Advanced Study Institute, Izmir, Turkey. June 26, 2000

[Lam89] Leslie Lamport, "A Simple Approach to Specifying Concurrent Systems", Communications of ACM, vol.32 no.1, pp32-45, January 1989

[Lohr92] Klaus-Peter Lohr, "Concurrency Annotations", Proc. on Object-oriented programming systems, languages, and applications, Canada, pp 327-340, 1992

[Miz99] M. Mizuno, "A structured approach for developing concurrent programs in Java", Information Processing Letters, Vol 69, No 5, pp232-238, 1999.

[Ober99] Ober. I., Stan. I "On the Concurrent Object Model of UML", Proceedings of EUROPAR' 99



[Omar03] Aldawud et.al., "UML Profile for Aspect-Oriented Software Development" Proc. 3rd Int. workshop on aspect oriented modeling, March, 2003

[OMG02] Object Management Group, "UML Profile for Schedulability, Performance, and Time Specification", OMG Adopted Specification ptc/02-03-02, July 1, 2002 (www.omg.org)

[Selic04] Selic, B., "On the Semantic Foundations of Standard UML 2.0", Lecture Notes in Computer Science vol. 3185, Springer-Verlag, 2004.

[SGW94] B. Selic, G. Gullekson, and P.Ward, "Real-Time Object-Oriented Modeling", John Wiley, New York, 1994

[WMC01] William E. McUmber, Betty H.C. Cheng, "A general framework for formalizing UML with formal languages", Proceedings of the 23rd International Conference on Software Engineering, Toronto, Canada, pp 433 – 442, 2001

APPENDIX: Formal Description of *cmUML* Semantics

In this section, we briefly outline the semantics definition approach for *cmUML* (as given in [1]). In general, defining the semantics of a language L (here *cmUML*) involves defining a mapping M between elements of L and concepts of chosen semantic domain S (here *symbolic transition systems* [11]). For this, a formal notation for *cmUML* specifications is required. The formal notation, a 10-tuple, defines the elements of *cmUML* as a type structure defining various primitive and complex types.

M = (T, Act, Att, Expr, F, E, P, S, C, I) where
T: A set of basic types and types for STATE, PORT, ENV and SERVICE classes (i.e. for 'Services')
Act: A finite set of UML *actions*
Att: A finite set of typed attributes of M
Expr: A finite set of expressions *expr* over Att
F: $F_t \cup F_p$ predefined operation types
E: A class of the type T_{ENV}, E = (e.Attr, e.Seq)
P: A class of the type T_{PORT}, P=(p.Attr, p.Seq, p.Acq)
S: A class of the type $T_{STATE,}$ S= (s.Att, s.Expr, s.Act, s.Ops, *Assign*, Q, Tr)
C: A finite, non-empty set of classes c, of the common super-type T_{OP} c = (c.Att, c.Param, c.Ret, c.Tf, Pre, Post, L)
I: A finite {<C, L_i, m!/m?/c, temp>} representing events to be sent/ received or condition to be met corresponding to each component C of type $t_C \in \{T_{PORT}, T_{OP}, T_{STATE}\}$

A symbolic transition system, say S, represents a type system of the necessary system variables subsuming the type system of *cmUML*. The behavioral semantics of a

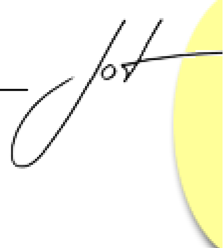
cmUML specification M is defined in terms of elementary transitions (or axioms) of the corresponding symbolic transition system (STS) $S_M \equiv (V_M, \theta_M, \rho_M, L_M)$ where V, the set of system variables, capture a dynamic execution of M, θ is initialization predicates, ρ the transition predicates, and L set of liveness axioms. A *snapshot* s of $S_M(V,\theta,\rho,L)$ represents the evaluation of the variables V by the underlying computational model with a synchronous global clock (where relevant variables are atomically updated by the end of discrete time intervals). The semantic description itself is divided in terms of various semantic modules corresponding to the main abstractions of *cmUML* framework (i.e. STATE, PORT, SERVICE). The module ENV represents an external environment scenario (useful for verification purpose). These modules define the operational semantics of objects of type T_{ENV}, T_{PORT}, T_{STATE}, $T_{SERVICE}$. Objects of type $T_{RESOURCE}$ are considered as passive objects which execute in caller's thread of control. The operational semantics of M is described in terms of execution of these semantic modules in terms of elementary atomic transitions specified intuitively in first order logic combined in imperative fashion. Concurrency semantics is captured by the system method '*fork*' as well as non-deterministic choices of the underlying computational model. We reproduce below the description of the semantic module representing dynamic configuration of an environment type.

Semantic Module *ENV-conf* :

```
ENV-status:  nowait, synch-wait
ENV-variables: status(ENV-status), in-queue(Queue),
         out-queue(Queue),msg(Msg-type),
         synch-msg(Msg-type);
ENVinit: status:=nowait, in-queue, out-queue, msg:= ε;
ENVasynch-send:msg=create(Msg-type: msg.type=call
                         ∧ msg.mode=asynch ∧……….);
                         msg.dest.in-queue.enqueue(msg);
ENVsynch-send:           status := synch-wait;
                         msg := create(Msg-type: msg.type=call ∧
                         msg.mode=synch ∧……….);
                         msg.dest.in-queue.enqueue(msg);
ENVasynch-receive:
                         msg := choose(Msg-type ∈out-queue);
                         if(msg != ε)   ENVlocal ∨ ENVnull;
ENVsynch-receive:  while (synch-msg = ε) wait;
              ENVlocal   ∨   ENVnull;   --do   local   actions/
nothing

ENVprocess:  while(true) {if status=synch-wait then
              ENVsynch-receive else ENVsynch-send ∨
              ENVasynch-send ∨ ENVasynch-receive ∨
              ENVlocal ∨ ENVnull; }
```

Detailed description of other semantic modules can be found in [JS06].



About the authors

Jagadish Suryadevara is a lecturer and a research scholar at Birla Institute of Technology and Science (BITS), Pilani, India. His areas of research interests are in UML based specification and analysis of concurrent systems, and real-time systems. He can be reached at jagadish@bits-pilani.ac.in.

Lawrence Chung is an associate professor in Erik Jonsson School of Engineering and Computer Science, University of Texas at Dallas, USA. Dr. Chung's research efforts are in the areas of Software Engineering, Requirements Engineering, Non-Functional Requirements, Software Architecture, Electronic Commerce/Business, Information Systems (Re-) Engineering.

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.