Towards Checking Protocol Conformance of Active Components

Youcef Hammal
Department of Computer Science, USTHB University, Algiers, Algeria
E-mail: hammal@lsi-usthb.dz

Abstract

As UML 2.x is now widely used by practitioners to document software architectures of concurrent real-time systems we propose an approach to make use of the UML protocol and behavior artifacts of components in order to achieve a verification activity at design time. We first discuss the issue of using protocol state machines to depict interactions of any active component with its environment through ports that specify provided and/or required interfaces. We propose to extend the use of such models to depict the flow of both asynchronous and synchronous events send not only from outside into the component but also in the other direction provided that the outgoing events affect the component state which is exposed to its environment via its ports. We then propose a formal method for conformance checking between component behavior and protocol state machines. These models are first mapped into abstract and flattened automata which are then combined and verified using model-checkers to uncover design flaws leading to deadlocks and race conditions in the implemented system. We explore also to what extent sequence diagrams could be suited over protocol state machines to annotate time constraints and how to use them for time consistency checking between implementation and specification models.

Keywords: *Protocol Conformance, Active components, UML Protocol state machine, Formal Methods for Component-Based Development, Reactive Systems.*

1. Introduction

We deal in this paper with the formal verification of reactive systems made up of distributed, interoperable, and substitutable components each of which encapsulates one or more services so that composite systems yield news services to the environment. Besides functional requirements, these systems that are typically concurrent and safety-critical must almost meet non-functional properties (e.g., temporal ones). Hence, such systems have to be specified and analyzed earlier (at design-time) to reduce the number of serious errors at later stages of development.

Moreover, to alleviate this challenging task, we propose to check such systems in a compositional way by combing and comparing each component behavior model with interaction protocol models of its ports and then proceed to their verification and analysis. In fact, protocol models allow specifying high-level requirements of any component according to a fragment-based approach contrary to the use of a single monolithic formal model which would capture the whole intended behavior. Such an approach allows the system developers to modularize their thinking and focus on each functionality in isolation. In addition, it is much easier to verify the behavior of the actual component against each fragment than comparing whole formal models.

We describe protocol and behavior models of components using UML 2.x notations as these are now widely used for documenting software architectures. In additions, UML 2 offers user-friendly and rich concepts and notations to depict behavioral and structural views of safety-critical and complex systems. However, some syntactic and semantics features remain ambiguous in UML2 component diagrams in view of modeling interaction protocols of highly reactive and time constrained components and particularly their ports which may use required as well as provided interfaces.

So, we address the issue of using protocol state machines for specifying interactions between any component and its environment through its ports, thereby depicting the flow of asynchronous as well as synchronous events send not only from outside into the component but also in the other direction. Hence, besides method call events incoming to a port and changing the component states one might consider also method invocations and signals that the component may require from its environment through this port provided that these outgoing events affect the component state (i.e., external view) which is exposed to its environment (clients or servers).

Our approach for conformance checking between component behavior and protocol state machines consists in abstracting these diagrams into a similar semantic domain, i.e., automata which are combined using various synchronization operators so that we could apply automated tools (i.e., model checkers) to verify their properties.

In this setting, we discuss how visual interpretations of state machines of active components could be misleading since these may do not conform to their formal dynamic semantics. Such misinterpretations result in design flaws leading to deadlocks and race conditions in the implemented system. We explore also to what extent sequence diagrams are suited over protocol state machines to annotate time constraints and how to make use of them for checking protocol conformance.

Besides our modular and automata-based approach, many approaches exploit other techniques for checking protocol conformance by components' implementations. For instance, authors of [2] use type-states for specifying architectural protocols and propose a static dataflow analysis for checking protocol conformance. In [4] authors use rewrite systems to capture the behavior of components and combine them with automata based protocols. By calculating the reachability of the fault state range one gets a counterexample violating properties specified by all protocols of the combined components. Authors of [10] propose an approach based on aspect oriented programming by converting protocol state machines into AspectJ code that can report any wrong call sequences in the implementation of Java components at runtime. Unfortunately, these techniques suffer from the state explosion problem when dealing with medium and large systems.

Lastly, the remainder of the paper is organized as follows: Section 2 summarizes the UML notations used to document software architectures. Section 3 presents UML state machines and explains how these are used to depict implementation behaviors and interaction protocols of components. Next, section 4 explains how these state machines are translated into abstract and flattened automata. Then, it addresses their combination and compatibility issues. Finally, a conclusion is given.

2. UML Component Diagrams

UML 2.x [9] provides a number of first-order concepts relevant to documenting software architectures such as component, interface, port and connectors.

UML specifies a component as a modular, encapsulated and autonomous unit that is replaceable within a system or subsystem. It has one or more provided and/or required well-defined interfaces (potentially exposed via ports), and its internals are hidden and inaccessible other than as provided by its interfaces.

As such, a component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant. Larger pieces of a system's functionality may be assembled by reusing components as parts in an encompassing component or assembly of components, and connecting them together.

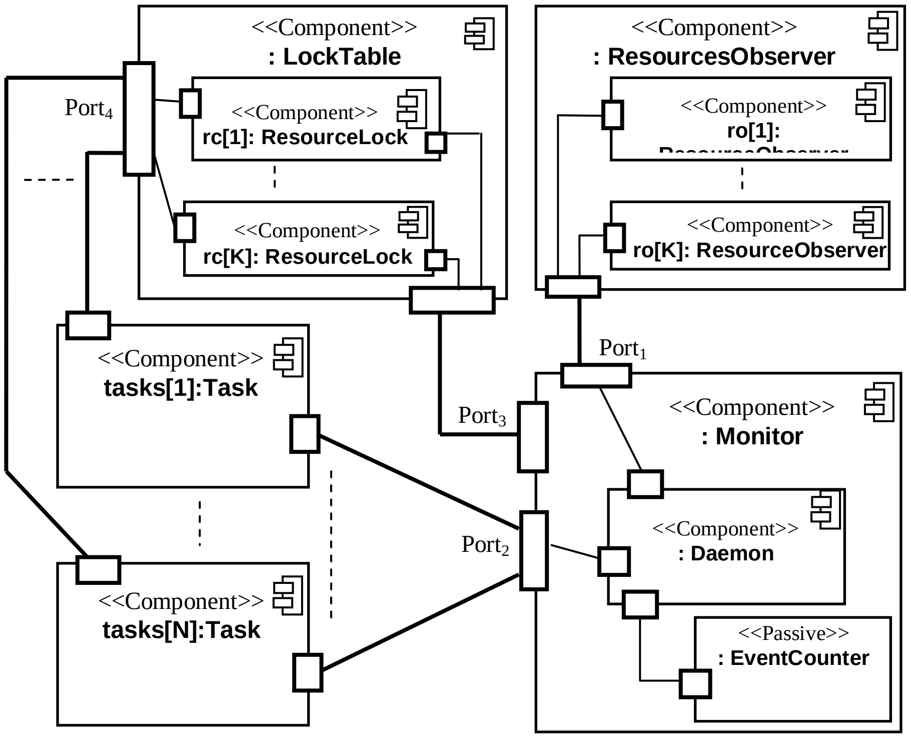


Figure 1. Composite Structure Diagram (ports are depicted by little rectangles attached to their components).

Optionally, a behavior such as a protocol state machine may be attached to an interface, port, and to the component itself, to define the external view more precisely by making dynamic constraints in the sequence of operation calls explicit. Other behaviors may also be associated with interfaces or connectors to define the "contract" between participants in a collaboration (e.g., in terms of activity, or interaction specifications).

A component also has an internal view (or "white-box" view) by means of its private properties and realizing internal classifiers [1] that may be displayed nested within the component shape. This view can be depicted using behavior state machines which are intended to realize the functionalities as exposed on the external view.

In order to manage complex structures of systems, two kinds of components are introduced that allow their hierarchical description: basic components and packaging components.

A basic component is defined as a specialized class having attributes and operations, and being able to participate in associations and generalizations. On the other hand, a packaging component is a structured classifier defined as a coherent group of elements. It extends the

[1] A classifier is an abstract model element that has a type such as class, component, etc.

concept of a basic component to formalize the aspects of a component as a 'building block' that may own a set of model elements.

Example. To explain our approach in the sequel, we present in Figure 1 the composite structure diagram of a reactive system inspired by that given in [5]. It consists of a set of concurrent processes (tasks) that compete for access to shared resources. To this end, they must first acquire access locks to resources before they can begin to manage them. In parallel, there is an observer agent for each resource in order to control its actual behavior and warns a monitor in case of the resource malfunction. If such a signal is received, the monitor scans all the observers and proceeds, for each one of them having sent an event, to disable the current task managing its resource and launches a recovery activity on the latter. Afterward, the monitor resumes the aborted task. However, if during this process some observes have raised events then the monitor has to achieve again the handling activity. Such situation is detected by comparing the numbers of events received at the beginning and the end of the polling process. If no new event is received then the monitor goes to its "*asleep*" state waiting for new events.

Note that when an instance of any component is created (e.g., Monitor), instances corresponding to each of its ports are created and held in the slots specified by the ports, in accordance with its multiplicity (e.g., *$Port_2$*). These instances are referred to as "interaction points" and provide unique references. A link from that instance to the instance of the owning component is created through which communication is forwarded to the instance of the owning component or through which the owning component communicates with its environment. It is, therefore, possible for an instance to differentiate between requests for the invocation of a behavioral feature targeted at its different ports. Similarly, it is possible to direct such requests at a port, and the requests will be routed as specified by the links corresponding to connectors attached to this port.

If connectors are attached to both the port when used on an internal element (e.g., Daemon) of a component (e.g., Monitor) and the port (e.g., *$Port_1$*) on the container of that internal element, the instance of the owning component will forward any requests arriving at this port along the link specified by those connectors.

3. State Machines as Behavior and Protocol Modeling Language

UML state machines [9] are finite state-transition systems supporting high-level concepts of hierarchy, orthogonality and abstraction. In addition to expressing the discrete behavior of the system components, state machines can also be used to express their usage protocol. These two kinds of state machines are referred to as behavioral state machines and protocol state machines. The former are used to specify behavior of various model elements (e.g., component and class instances) whereas the latter are used to express usage protocols.

3.1. Behavior State Machines

A behavior state machine [9] shows "a behavior that specifies the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions". An event can be a signal, an operation invocation, a time passage or a condition change, whereas a state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity or waits for some event. Here, an event is an occurrence of stimulus (originating from either the environment or the object itself) that can trigger a state transition whose action performance may raise new events.

As expected, the semantics of event processing is based on the run-to-completion assumption which means that an event can only be dequeued and dispatched if the processing of the previous current event is fully completed. The processing of a single event is referred to as a run-to-completion step which represents a passage between two stable state configurations of the state machine. Hence, before launching a run-to-completion step, a state machine should be in a stable status with all actions (but not necessarily activities) completed.

During a transition, a number of actions may be executed. If such an action is a synchronous operation invocation on an instance(s) executing a state machine, then the transition step is not completed until the invoked object(s) complete its run-to-completion step.

For instance, Figure 2 illustrates the behavior state diagram of the monitor component. It contains two communicating state machines related to its constituents where that of the passive object "EventCounter" is simple (since it only responds to method calls) while the state machine of "Daemon" encompasses high-level features of active components like orthogonal regions and transitions triggering asynchronous signals and invoking methods on its own from other components. Here also, the simple term "current state" can be quite confusing because more than one state can be active at once (due to concurrent states). So, we refer to such an active state configuration as status.

The behavior state machine in Figure 3 shows how a task acquires a lock over a resource and manages it until it releases its lock or receives an “abort” signal. In this last case, the task has to release also the lock and then stays aborted until it receives a “resume” order.

Furthermore, Figure 4 contains two behavior state machines: The first one is related to one resource lock subcomponent which receives and checks requests from tasks. Once some task gets the lock over this resource the latter remains locked until it receives a release signal from the owner task. However, the state machine has to handle every request for the owner identifier whatever its current state. The second state machine is related to the observer agent of one resource, which has to continually check the status of this resource and alerts the monitor in case of any abnormal function. The resource observer shall return to its initial state after some recovery action is achieved. The agent is able as well, to handle requests for notifying the resource current status (0 for normal status or 1 for abnormal status).

Every time a malfunction signal is received, the monitor scans all the observers (as specified in Figure 2). For each resource whose the observer sent such a signal, the monitor sends an abort signal to its owner task and launches a recovery activity on the latter. Afterward, the monitor resumes the aborted task. However, if other observers raise malfunction events during this process then the monitor has to achieve again the handling activity. Such situation is detected by comparing the numbers of events received respectively at the beginning and at the end of the polling process. If no new event is received then the monitor goes to its "*asleep*" state waiting for new events.

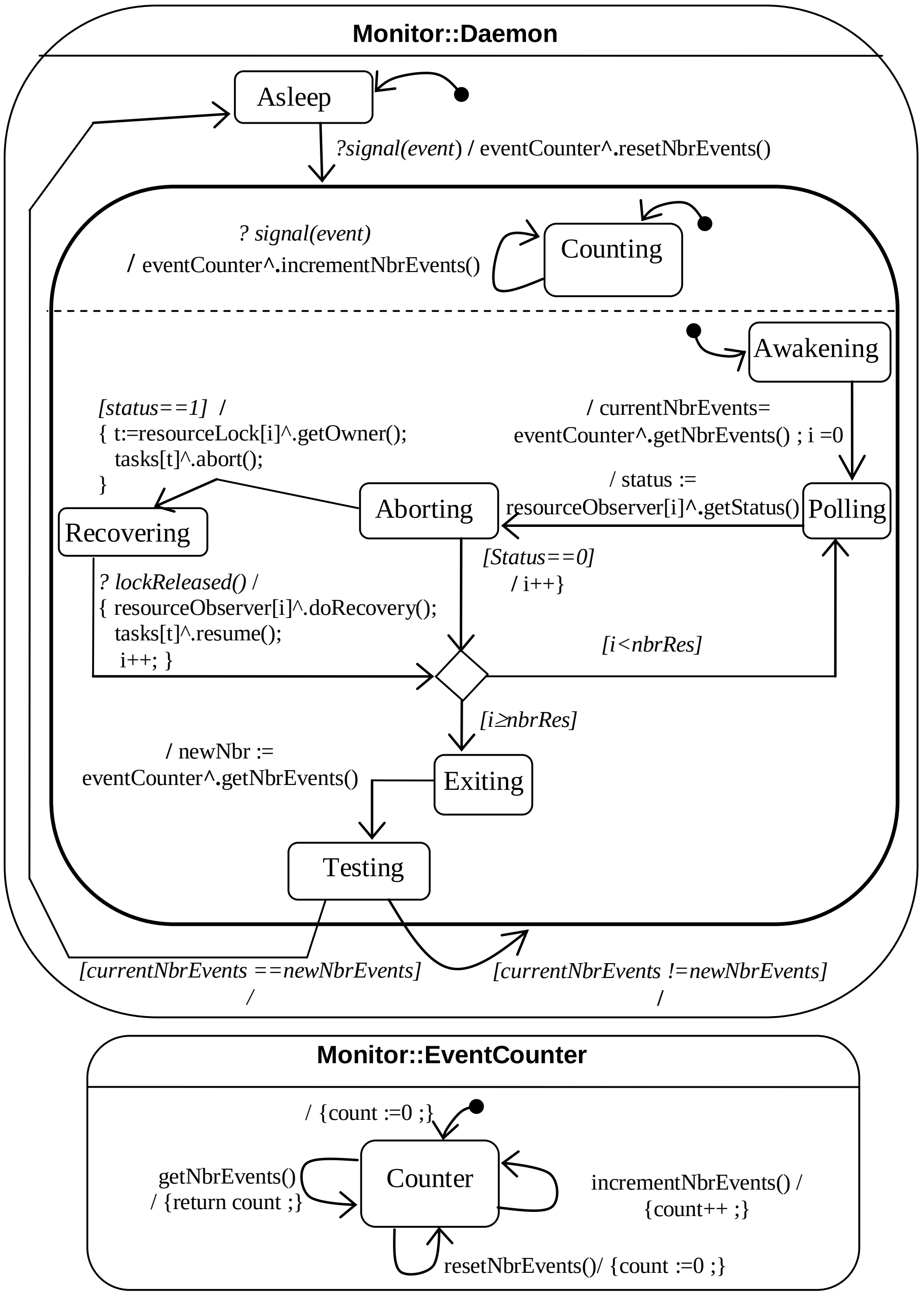


Figure 2. Behavior State Machines of the Monitor Subcomponents.

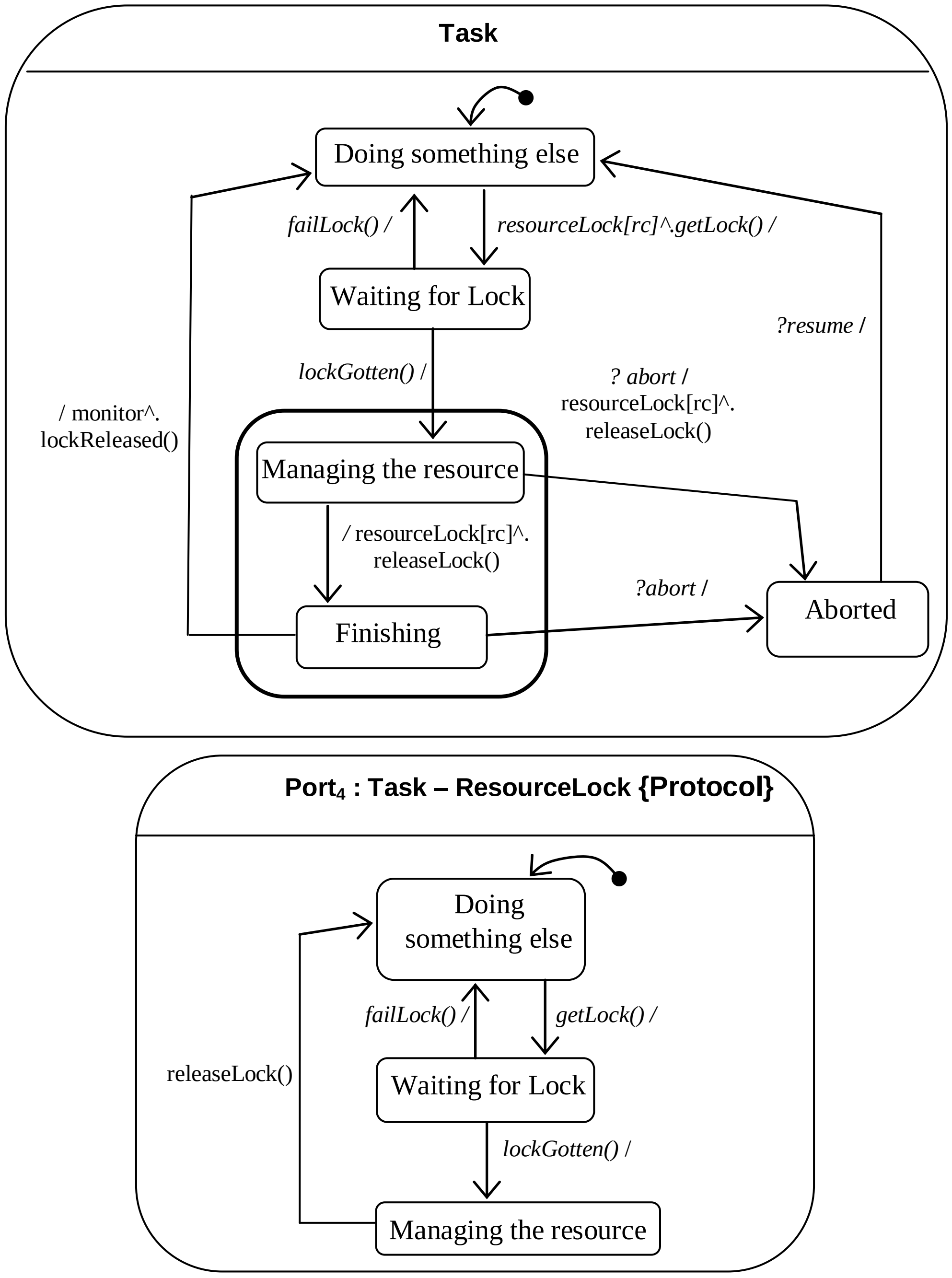
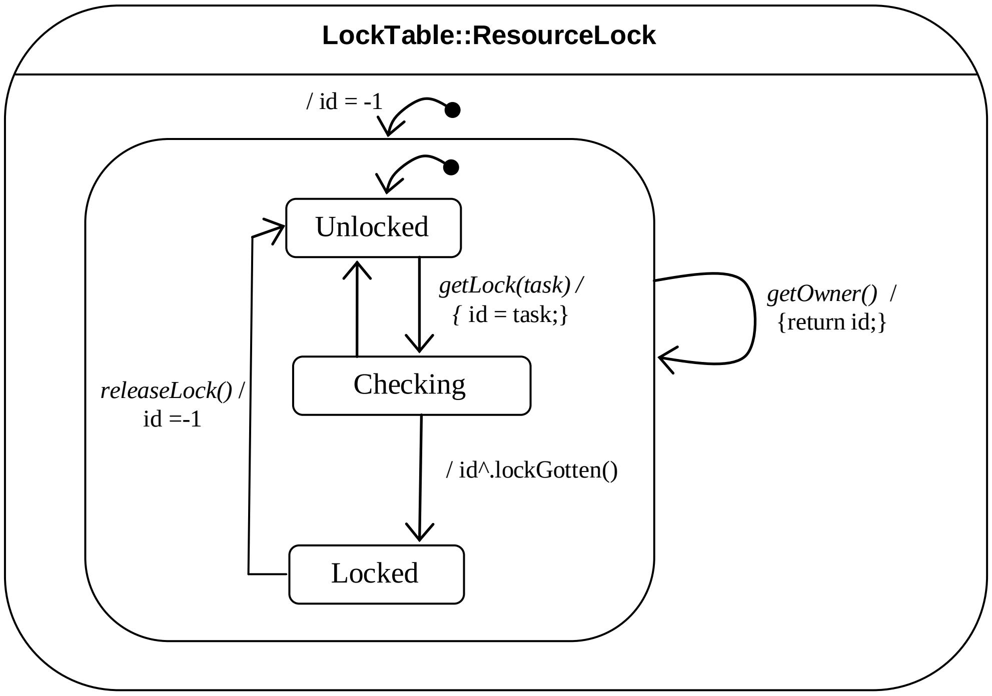


Figure 3. Behavior and Protocol State Machines of a Task Component.



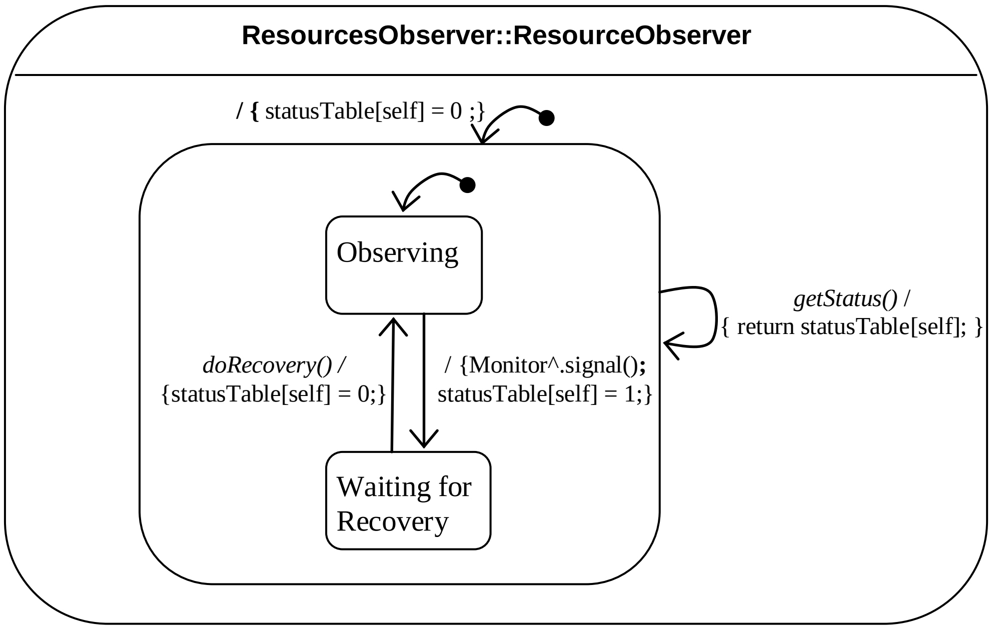


Figure 4. Behavior State Machines of Resource Lock and Observer Components.

3.2. Protocol State Machine

A protocol state machine [9] specifies which operations of the component can be called in which state and under which condition, thus specifying the allowed call sequences on the component's operations. In this way, an instance lifecycle can be created for a component, by specifying the order in which the operations can be activated and the states through which an instance progresses during its existence. Note that because protocol state machines do not preclude any specific behavioral implementation, and enforces legal usage scenarios of classifiers, interfaces and ports can be associated to this kind of state machines. For instance, protocol state machines in Figure 5 and Figure 6 are assigned to the ports of the monitor and task components, respectively.

The protocol state machine must represent all operations that can generate a given change of state for a component. Those operations that do not generate a transition are not represented in the protocol state machine. The protocol states present an external view of the component that is exposed to its clients. Depending on the context, protocol states can correspond to the internal states of the instances as expressed by behavioral state machines, or they can be different. If two ports are connected, then the protocol state machine of the required interface (if defined) must be conformant to the protocol state machine of the provided interface (if defined).

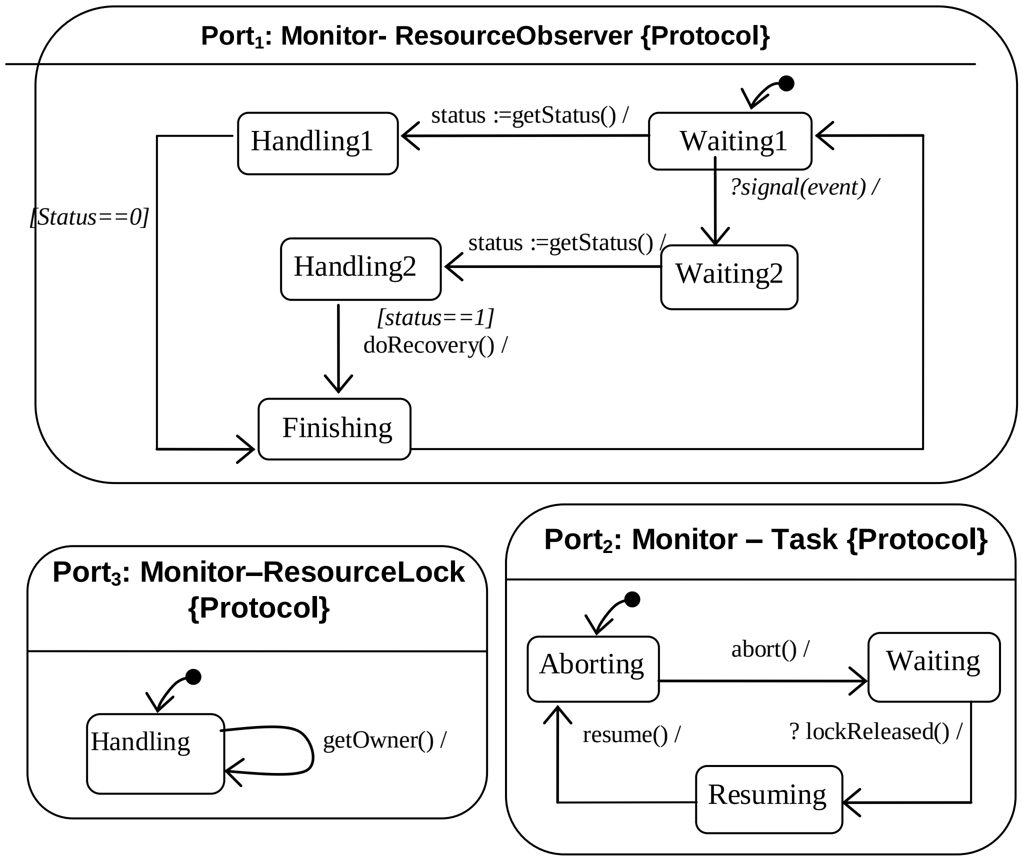


Figure 5. Protocol State Machines of the Monitor Ports.

Protocol states cannot have entry, exit, or do activity actions. Protocol state machines can use compound transitions, sub-state machines, composite states, and concurrent regions but cannot have deep or shallow history pseudo-states.

A protocol transition specifies a legal transition for an operation. It has the following information: a pre-condition (guard), a trigger, and a post-condition. Every protocol transition is associated to zero or one operation that belongs to the component. The protocol transition specifies that the associated (referred) operation can be called for an instance in the origin state under the initial condition (guard), and that at the end of the transition, the destination state will be reached under the final condition (post).

The effect action is never specified. It is implicit, when the transition has a call trigger: the effect action will be the operation specified by the call trigger. It is unspecified in the other cases, where the transition only defines that a given event can be received under a specific state and pre-condition, and that a transition will lead to another state under a specific post-condition, whatever action will be made through this transition.

3.3. Discussion on Protocol Transitions

Apart from the operation call event, events are generally used for expressing a dynamic behavior interpretation of protocol state machines. An event that is not a call event (e.g., signal event) can be specified on protocol transitions. In this case, this specification is a requirement from the environment external to the state machine: it is legal to send this event to an instance of the component only under the conditions specified by the protocol state machine.

Note that a signal triggers a reaction in the receiver in an asynchronous way and without a reply. The sender of a signal will not block waiting for a reply but continue execution immediately. The related signal event occurs when the signal message, originally caused by a send action executed by some object, is received by another (possibly the same) object. The signal event may cause a response, such as a state machine transition as specified in the classifier behavior of the receiver object.

We recall that we are interested in use of protocol state machines to depict legal transitions that active components can trigger. As such components have their own thread of control they are able not only to react to external stimuli but also to trigger, on their own, method calls and signals conveyed to their environment and change their states.

Furthermore, labels on the protocol transitions do not only depict operations from its provided interfaces (those that are claimed by its clients) but also operations from its required interfaces. In the former case, operations are triggered from the component clients whereas in the latter case operations are internally invoked to claim services from its environment.

As illustrated on Figure 5, labels of Monitor transitions are method calls or signal events that are triggered either by the environment or by the component itself. However, both of them lead to changing the component status.

3.4. Abstract Representation of Events and Actions

As the rationale behind our formalization process is the rigorous analysis of the resulting behavioral models, we consider state machines at a more abstract level and focus on their observable behaviors. In this respect, we use our translation method given in [6] to map state charts into Petri nets whose state spaces can be explored to yield automata based models.

In fact, we are interested through this process in extracting events encompassed by labels of transitions on the state machine, namely triggering (i.e., received) events and triggered events that are generated (i.e., send) by their raised actions.

Once applying the translation process to state machines of the monitoring system, we get the resulting automata (illustrated by figures from 6 to 11) which depict behavior and protocol interaction of the monitoring system components. These automata have been manually enhanced in such a way that we can check them using UPPAAL tool [11].

Note that, at an abstract level of specification, one might mainly observe instantaneous events generated or consumed by components and their temporal order for analysis purposes. Therefore, we use the process algebra concept of communication gate supporting aforementioned basic events in such a way we could denote them with one of the two following classical forms (see Figure 6 and Figure 7):

- ✓ g! denotes an event sending. If the event is a method call, the related action blocks up until a result is returned, while an asynchronous signal event is not blocking.
- ✓ g? denotes an event receipt which is always a blocking one.

Moreover, gates may be either unary, binary or multi-way points of communication:

- ✓ A unary gate is visible only within its owner component. So, the gate is known and used only inside this component to carry out internal operations.
- ✓ A binary gate is a rendezvous gate between exactly two components. Once these two components are combined, this gate is hidden to their environment so that no interaction with a third component may happen on it. For instance, method invocations always use this kind of gate to address a service request to a required component (server object).
- ✓ A multi-way gate is a broadcasting gate. It remains visible even after using it to handle communications between many subcomponents. As a result, every signal we broadcast on this kind of gate can be captured by other components having access to it.

The next step to enable the translation of our high-level state machines into more abstract automata is to give below the representation schemes of the state machine events and actions into abstract events to put upon transitions of the target automata:

A. Method Calls from the environment are input (i.e., receipt) events that appear only as triggering events on the state machine transitions (both behavior and protocol ones). The associated operations implement services that the component offers to its environment. These events are denoted as follows in our automata:
componentName.methodName [componentInstanceReference]?.
If there is one instance of this component and no other component owns a method with the same name, this notation can be simplified to: *methodName?.*
For instance, many arcs in the resource observer automaton (see Figure 10) are labeled with input events (method calls) such as "getStatus[rc]?" and "doRecovery[rc]?" where "rc" is the resource reference. Similarly, the resource lock automaton (see Figure 11) contains many arcs labeled with methods calls such that "getOwner[rc]?", "getLock[rc]?" and "releaseLock[rc]?".

B. Method Calls that the component sends to its environment are related to its required services. Such events are output (i.e., send) events which result from the execution of either actions on some triggered transitions of its behavior state machine or operations specified on protocol transitions that the component is allowed to trigger to require some service from its environment (namely, a server component). We thus denote them as follows: *serverComponentName.methodName [serverInstanceReference]!.*
In the same way, when there is only one instance of the server component and no other component offers a method having the same name, this notation can be simplified to: *methodName!.*

For instance, the monitor automaton (see Figure 6) raises many method calls towards the other components during the polling process such as “getStatus[i]!”, “getOwner[i]!” and “doRecovery[i]!” where “i” is some resource reference. Similarly, the task automaton in Figure 8 contains many arcs labeled with methods calls such as “getLock[r]!” and “releaseLock[r]!”.

C. For signals, we need more information about sender and receiver entities since these can be broadcast unlike method calls. A signal event receipt by the component (as specified by any of its provided interfaces) is an input event denoted as follows:
SignalName[ReceiverInstanceReference,SenderInstanceReference]?
If one instance exists for any of the participants, this can be eliminated from the gate name simplifying thus the notation. If both of them are discarded the event is then denoted by: *signalName?*.
Examples of this kind of events are “signal[e]?” and “lockReleased[t]?” in the monitor automaton (Figure 6) where “e” and “t” represent, respectively, their sending entities. Similarly, the events “lockGotten[rc]?” and “failLock[rc]?” in the task automaton (Figure 8) are also receipt signals.

D. Similarly, signal event send from the component (as specified by any required interface) is an output event denoted as follows:
SignalName[ReceiverInstanceReference,SenderInstanceReference]!.
If no naming conflicts exist this notation can be rewritten to: *signalName!*.
In Figure 8, the event “lockReleased[id]!” is a signal that the task sends to the monitor. Moreover, the resource lock automaton (Figure 11) warns the owner task by sending the events “lockGotten[rc]!” and “failLock[rc]!”.

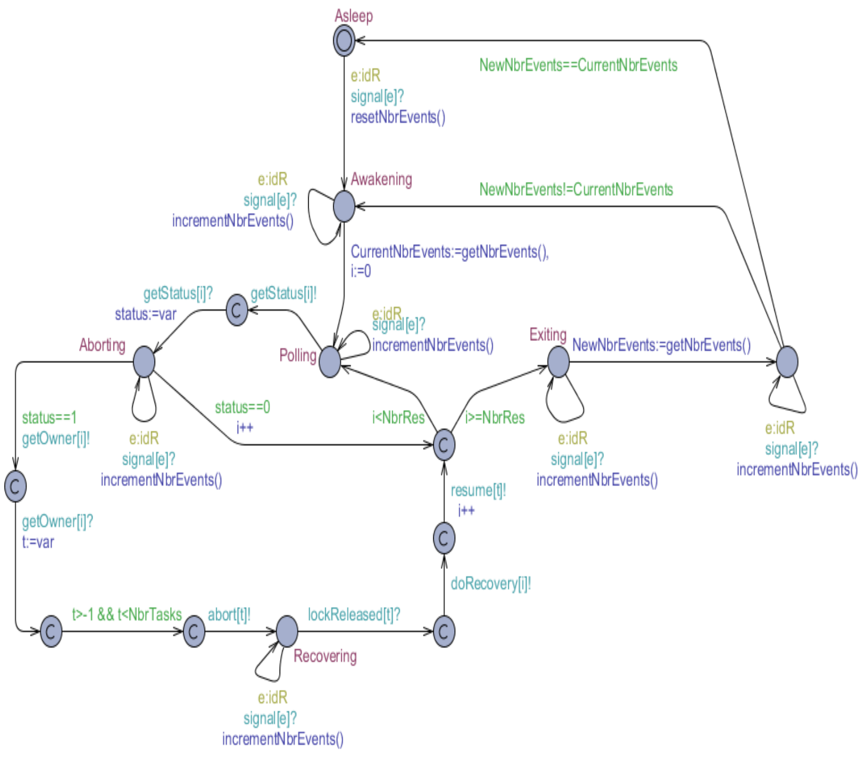
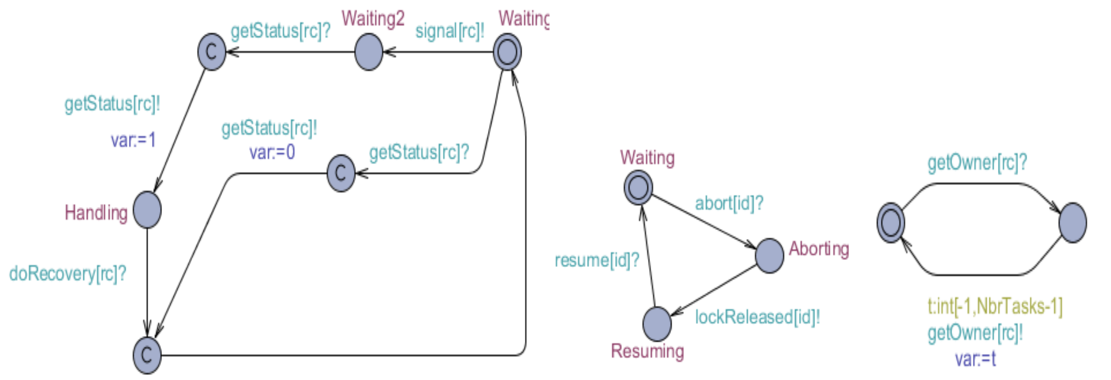


Figure 6. Automaton of the Monitor Component.



Port$_1$ Monitor-Resource Observer *Port$_2$ Monitor-Task* *Port$_3$ Monitor-ResourceLock*

Figure 7. Automata of Protocol State Machines of the Monitor Ports.

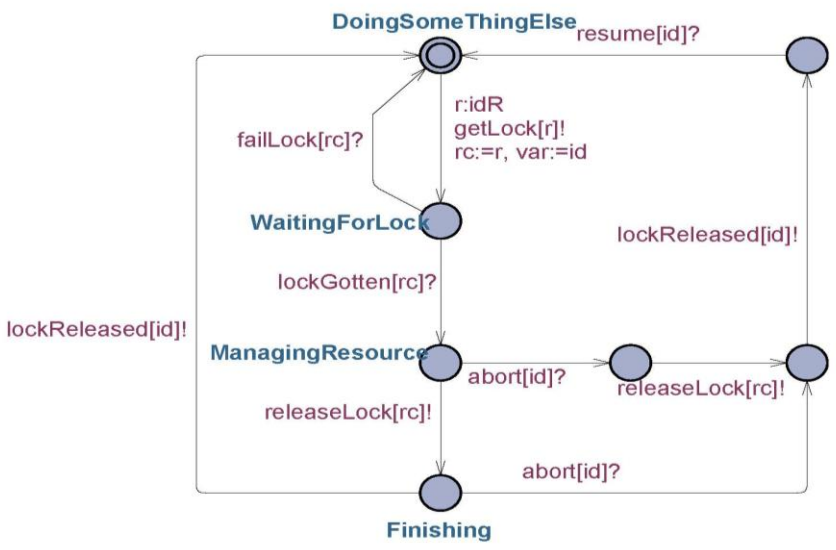
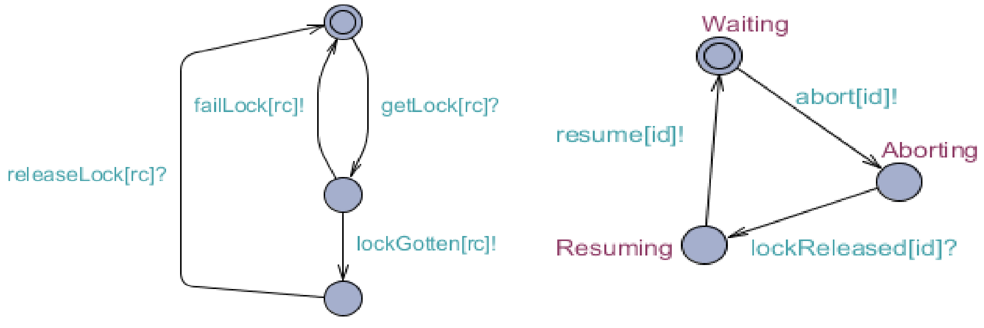


Figure 8. Automaton of a Task Component.



Port$_4$ Task-ResourceLock *Port$_5$ Task-Monitor*

Figure 9. Automata of Protocol State Machines of a Task Ports.

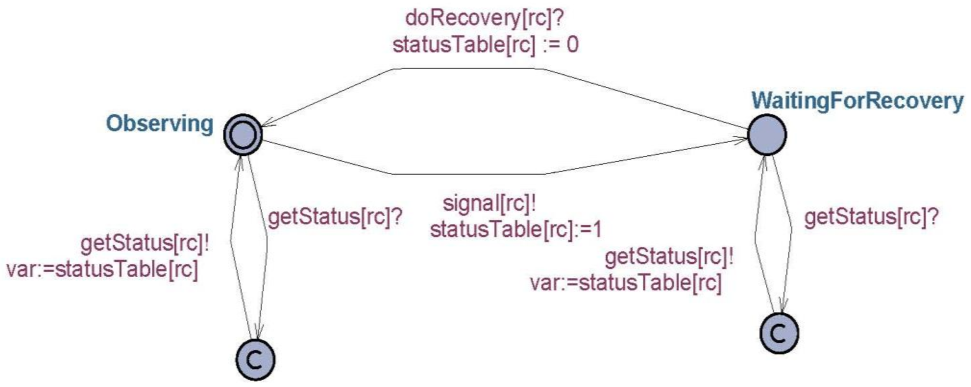


Figure 10. Automaton of a Resource Observer.

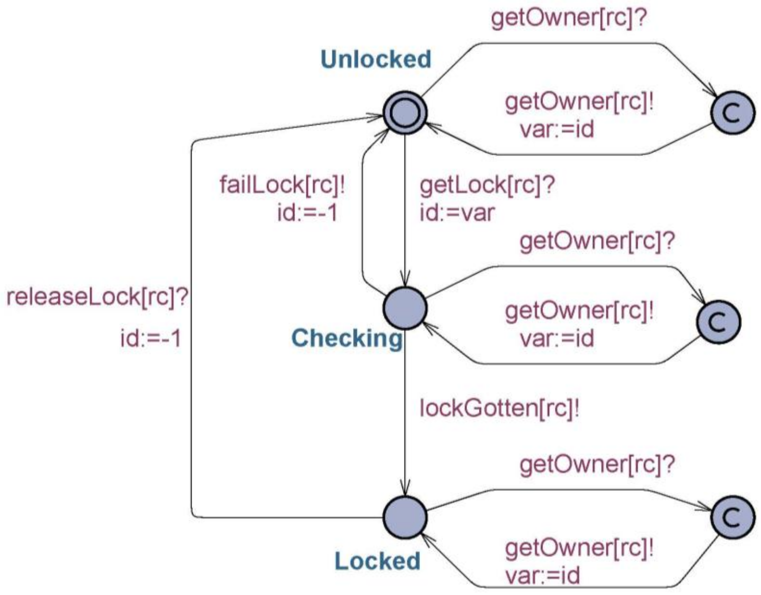


Figure 11. Automaton of a Resource Lock.

4. Protocol Conformance Checking

Reactive systems are continuous systems with various kinds of concurrent components evolving under time constraints. Hence, any possible design artifacts or behavioral models of their implementation are supposed to be free from sink states which depict deadlock configurations. However, ensuring this safety property is a hard task because most of automata based models adopt the assumption that components should simply reject events offered by their environment if their current statuses do not allow them to accept and handle these events. Therefore, some ill-conceived synchronization cases may raise deadlocks. this situation can be more complicated if we consider also the time deadlines that actions involved in the synchronization have to meet [1,3].

Let $\boldsymbol{A}_{Impl}$ be the component implementation automaton issued from its behavior state machine, and let $\boldsymbol{A}_{Port_k}$ be its specification automaton issued from the protocol state machine of the k^{th} port (using the translation method given in [6]).

The component description is the behavioral automaton obtained by combination of the two automata, namely the implementation model depicting the design artifacts and the specification model illustrating its intended interactions with its environment through some port. Thus, the component combined automaton is: $\boldsymbol{A} = \boldsymbol{A}_{Impl} \otimes \boldsymbol{A}_{Spec}$ where $\otimes$ is a parallel product which compels the two graphs to synchronize for the performance of common actions and lets them achieve uncommon actions in an interleaved way. The choice of the merging operator depends on the type of synchronization operator of the checking tool one would like to use. In the following, we present two merging patterns: the first one uses an CSP-like operator $\otimes_1$ that the "LTSA" tool adopts [8] whereas the second pattern uses a CCS-like operator $\otimes_2$ that UPPAAL tool adopts [11].

4.1. Automata Composition using CSP Synchronization Operator

Before giving the merging operator of implementation and specification automata, we present below the constituents of these automata:

$\boldsymbol{A}_{Impl} = < Q_I, \Sigma_I, \rightarrow_I, q_I^0 >$ where:

- Q_I is the set of states of the component implementation automaton (or the graph nodes as illustrated in Figure 6) that we get from state exploration of its behavior state machine,
- Σ_I is the set of events extracted from labels on transitions of the behavior state machine. They are either triggering or triggered events,
- $\rightarrow_I \subseteq Q_I \times \Sigma_I \times Q_I$ is the set of automaton transitions related to moves between configuration statuses of the behavior state machine (in other terms, these are transitions between markings of the equivalent Petri net),
- q_I^0 is the initial state (i.e., node) of the automaton.

Similarly, the specification automaton related to a port $Port_k$ has the following form:

$\boldsymbol{A}_{Port_k} = < Q_k, \Sigma_k, \rightarrow_k, q_k^0 >$ where*:*

- Q_k is the set of states of the component as depicted on the protocol state machine related to the k^{th} port (these states are illustrated by the graph nodes in Figure 7),
- Σ_k is the set of all allowed interactions in which the component can be involved with its environment through the k^{th} port. Such interactions are extracted from the protocol transitions showing the method calls and signals that our component could receive or raise provided that these events change its state exposed to its environment. Note that we have always: $\Sigma_k \subseteq \Sigma_I$.
- $\rightarrow_k \subseteq Q_k \times \Sigma_k \times Q_k$ is the set of transitions between configuration statuses that we compute from the protocol state machine,
- q_k^0 is the initial state (i.e., node) of the automaton,

We give below the merging operator based on the parallel operator of the CSP process algebra [3].

Definition 1. *A merging product* $\otimes_1$ *of two graphs* $\boldsymbol{A}_{Impl}$ *and* $\boldsymbol{A}_{Port_k}$ *yields a new automaton:*

$\boldsymbol{A} = < Q, \Sigma, \rightarrow, q_0 >$ *such that:*

- $Q = Q_I \times Q_k$ *with* $q_0 = (q_I^0, q_k^0)$.
- $\Sigma = \Sigma_I$ *is the set of actions the automaton* $\boldsymbol{A}$ *performs either in individual or cooperative way.*
- $\rightarrow = \{(p_I,p_k) \xrightarrow{\quad a \quad} (q_I,q_k) \mid a \in (\Sigma_I \cap \Sigma_k) \wedge (p_I \xrightarrow{\quad a \quad} q_I) \in \rightarrow_I \wedge (p_k \xrightarrow{\quad a \quad} q_k) \in \rightarrow_k\} \cup \{(p_I,p_k) \xrightarrow{\quad a \quad} (q_I,q_k) \mid a \in (\Sigma_I \backslash \Sigma_k) \wedge (p_I \xrightarrow{\quad a \quad} q_I) \in \rightarrow_I \wedge (p_k = q_k)\}$

Now, to compare automata and check their conformance, we need an observational criterion such as the strong bisimulation [3]:

Let G_1 and G_2 be two labeled transition systems (also referred to as graphs for more conciseness): $G_i = \langle Q_i, \Sigma_i, \rightarrow_i, q_i^0 \rangle$ for i=1,2 and let Σ denote the set $\Sigma_1 \cup \Sigma_2$.

Definition2. A (strong) bisimulation is a binary and reflexive relation $\Re \subseteq Q_1 \times Q_2$, such that for every $(p,q) \in Q_1 \times Q_2$, $p \Re q$ iff: $\forall a \in \Sigma, \forall p' \in Q_1: p \xrightarrow{a} p' \Rightarrow \exists q' \in Q_2: q \xrightarrow{a} q' \wedge p' \Re q'$.

Two graphs G_1 and G_2 are bisimilar ($G_1 \approx G_2$) when their initial nodes are strongly bisimilar ($q_1^0 \approx q_2^0$).

Definition 3. *We say that an implementation* $\mathbf{A}_{Impl}$ *fulfills its protocol specification via the* k^{th} *port* $\mathbf{A}_{Port_k}$ *(note it* $\mathbf{A}_{Impl} \models\!\!= \mathbf{A}_{Port_k}$*) if the component combined description and implementation automata are bisimilar, i.e.,* $\mathbf{A}_{Impl} \approx \mathbf{A}_{Impl} \otimes_1 \mathbf{A}_{Port_k}$.

This definition ensures that each state of the protocol statechart can be simulated by some of the states of the behavioral state machine.

Definition 4. *We say that an implementation* $\mathbf{A}_{Impl}$ *fulfills its protocol specification via all its M ports* $\{ \mathbf{A}_{Port_k} \}_{k=1..M}$ *(*$\mathbf{A}_{Impl} \models\!\!= Spec_{Ports}$*) if the implementation automaton is bisimilar to the automaton generated from the combination of this latter with all ports automata, i.e.,* $\mathbf{A}_{Impl} \approx \mathbf{A}_{Impl} \otimes_1 (\otimes_{1k=1..M} \mathbf{A}_{Port_k})$.

Proposition 1. $\forall k \in \{1..M\}: \mathbf{A}_{Impl} \models\!\!= \mathbf{A}_{Port_k} \Rightarrow \mathbf{A}_{Impl} \models\!\!= Spec_{Ports}$.

Proof. We have $\forall k \in \{1..M\}: \mathbf{A}_{Impl} \models\!\!= \mathbf{A}_{Port_k}$. We prove by induction on the number of ports k that $\mathbf{A}_{Impl} \approx \mathbf{A}_{Impl} \otimes_1 (\otimes_{1k=1..M} \mathbf{A}_{Port_k})$. For $k=1$ we know, by the premise of the implication, that the property is true for each port.

We suppose now that the property is true up to $k=i$ and prove it remains true for $k=i+1$. Since $\forall k_1, k_2 \in \{1..M\}$ such that $k_1 \neq k_2$, we have $\Sigma_{k1} \cap \Sigma_{k2} = \varnothing$ and since $\forall k \in \{1..M\}$: $\mathbf{A}_{Impl} \models\!\!= \mathbf{A}_{Port_k}$, we can easily show that that $\mathbf{A}_{Impl} \approx \mathbf{A}_{Impl} \otimes_1 (\otimes_{1k=1..i+1} \mathbf{A}_{Port_k})$.

4.2. Automata Composition using CCS Synchronization Operator

In order to use the UPPAAL tool and its CCS-like synchronization operator $\otimes_2$, we adapt the protocol automaton to enable its synchronization with implementation automaton according to rules of table 1. To this end, we simply transform the label a of each transition on the protocol automaton as follows:

Wherever the label has the form "*g?*" then it is changed into "*g!*" and vice versa.

Table1. Inference Rules for Synchronization of Transitions

R_1:	$\dfrac{(p_I,a,q_I) \in \rightarrow_I, (p_k,b,q_k) \in \rightarrow_k, (a=g! \wedge b=g?) \vee (a=g? \wedge b=g!)}{((p_I,p_k), \tau, (q_I,q_k)) \in \rightarrow}$
R_2:	$\dfrac{(p_I,a,q_I) \in \rightarrow_I, (p_k,b,q_k) \notin \rightarrow_k, (a=g! \wedge b=g?),\ g \text{ unary or broadcast gate}}{((p_I,p_k), \tau, (q_I,p_k)) \in \rightarrow}$

Since the CCS synchronization operator $\otimes_2$ is binary and yields τ-actions, we can check the conformance of the component behavior against its ports' protocols by checking the deadlock-freeness of the synchronized product via $\otimes_2$ between its implementation graph with the combination of all its ports' graphs by the CSP-like operator $\otimes_1$.

Definition 5. *We say that an implementation $\boldsymbol{A}_{Impl}$ fulfills (denoted $|{=}{=}{=}_{CCS}$) its protocol specification via all its M ports $\{\boldsymbol{A}_{Port_k}\}_{k=1..M}$ ($\boldsymbol{A}_{Impl}$ $|{=}{=}{=}Spec_{Ports}$) if the combined automaton $\boldsymbol{A}_{Impl} \otimes_2 (\otimes_{1k=1..M} \boldsymbol{A}_{Port_k})$ is deadlock free.*

We can easily, prove that Proposition 1 is preserved in this context, as follows.

Proposition 2. $\forall k \in \{1..M\}$: $\boldsymbol{A}_{Impl} \; |{=}{=}{=}_{CCS} \boldsymbol{A}_{Port_k} \Rightarrow \boldsymbol{A}_{Impl} \; |{=}{=}{=}_{CCS} Spec_{Ports}$.

Unfortunately, the previous definitions state only how to check that the component implementation behaves according to the allowed sequences of operations as specified by protocol models. This means that the component may interact with its environment without deadlocks but liveness properties could be not guaranteed. Hence, we promote use of model checkers to uncover design flaws (along with theorem provers to overcome state explosion problem). For instance, we have checked using UPPAAL the following properties among others (which allow us to correct the related behavior state machines):

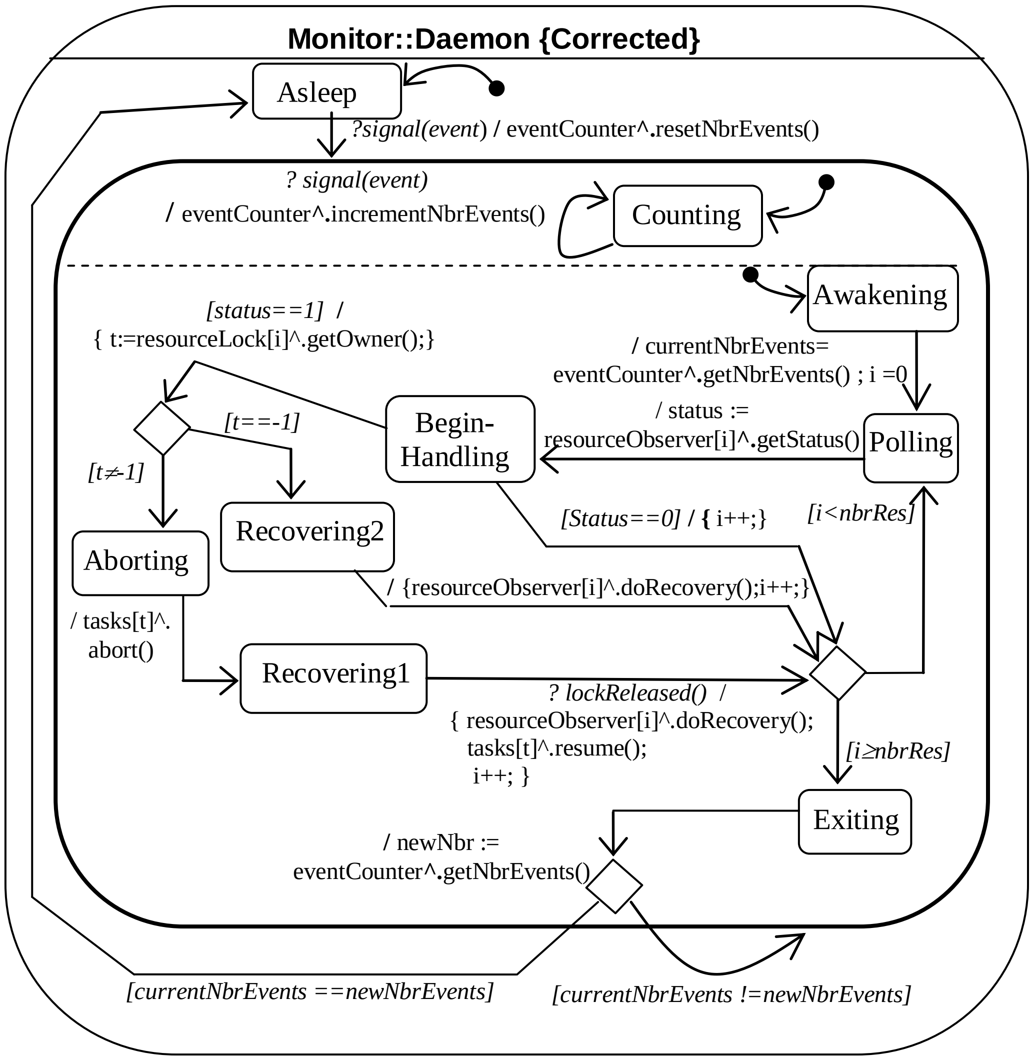

Monitor::Daemon {Corrected}
Asleep
?signal(event) / eventCounter^.resetNbrEvents()
? signal(event)
/ eventCounter^.incrementNbrEvents()
Counting
Awakening
[status==1] /
{ t:=resourceLock[i]^.getOwner();}
/ currentNbrEvents=
eventCounter^.getNbrEvents() ; i =0
[t==-1]
Begin-Handling
/ status :=
resourceObserver[i]^.getStatus()
Polling
[t≠-1]
Recovering2
[Status==0] / { i++;}
[i<nbrRes]
Aborting
/ {resourceObserver[i]^.doRecovery();i++;}
/ tasks[t]^.
abort()
Recovering1
? lockReleased() /
{ resourceObserver[i]^.doRecovery();
tasks[t]^.resume();
i++; }
[i≥nbrRes]
Exiting
/ newNbr :=
eventCounter^.getNbrEvents()
[currentNbrEvents ==newNbrEvents]
[currentNbrEvents !=newNbrEvents]

Figure 12. The Corrected Behavior State Machine of the Monitor.

- Checking the absence of deadlock allowed us to uncover a design error in the behavior state machine of Figure 2. Indeed, we assumed that the method call of "getOwner" at the state "Aborting" returns always the identifier of the task managing the resource we want to recover. But, a malfunction signal may be raised from an unlocked resource so that the returned value would be invalid for the next operations. For that reason, the model checking of the monitor component model (only the monitor along with its ports) using UPPAAL asserts that the two following CTL[2] formulae are not satisfied:
    - ✓ A[](Monitor.BeginHandling imply Monitor.Aborting),
    - ✓ A[] not deadlock.
- We checked that all signals send from resource observers during the monitor activity are always handled before this one returns to its "asleep" state. The corresponding CTL formula checked is:
    - ✓ A[] (Monitor.Asleep imply forall (k:idR) Port1(k).Waiting1).

  The model checking asserts that the property is not satisfied because the receipt of signals in the state machine of Figure 2 is achieved in an orthogonal region. According to the semantics of UML, the related events may occur at all stable statuses including "Exiting" and "Testing".
  Considering the two aforementioned flaws, the behavior state machine of the monitor has been corrected into the one depicted in Figure 12 as follows: We merge the two states "Exiting" and "Testing" and their two outgoing transitions using a dynamic choice pseudo-state whose guards are evaluated dynamically after achieving the associated action. Moreover, the states “Aborting” and “Recovering” have been refined in order to correctly handle all possible task identifiers (including -1 depicting a situation of an unlocked resource whose observer has sent a malfunction signal). The corresponding automaton of the revised state machine is given in Figure 13.
- We successfully checked as well, that tasks release their locks over resources once they finish their handling or when they are aborted by the monitor.

[2] CTL (Computation Tree Logic) is a temporal logic for specifying properties to check with UPPAAL.

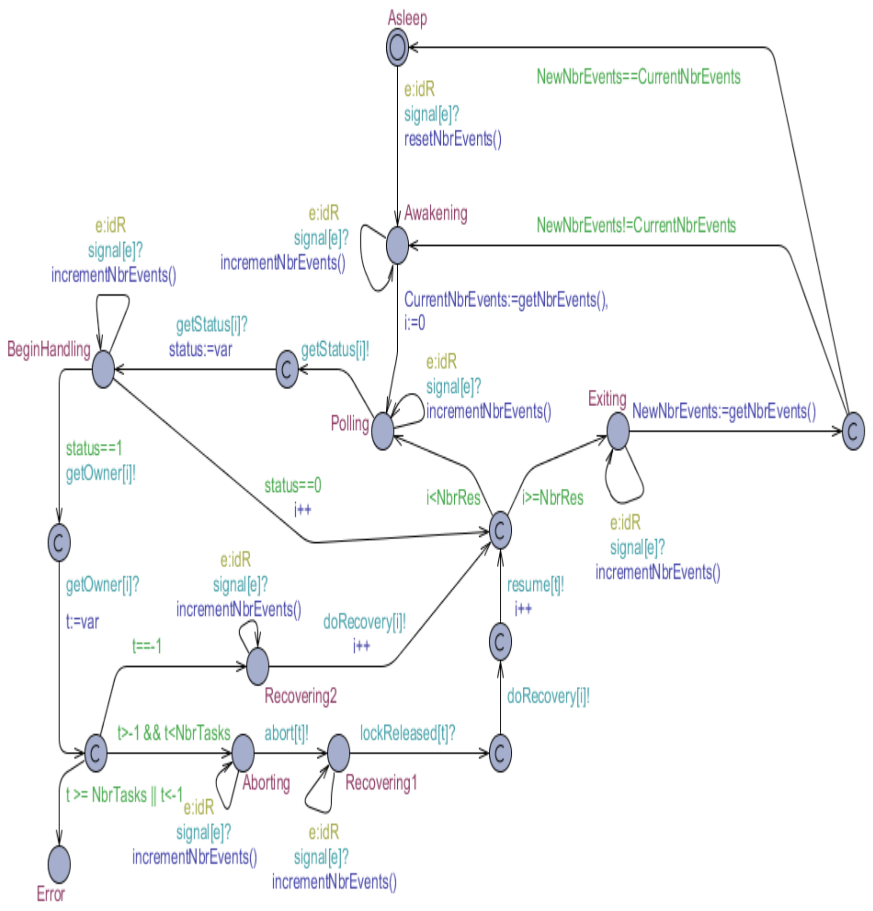


Figure 13. Automaton of the Corrected Monitor State Machine.

4.3. Timed Protocol Conformance Checking

Timed automata [1] are very suited to illustrate timed computations that any system or component should perform thanks to the use of logical clocks and their time constraints. Moreover, many analysis techniques and tools are available for this kind of automata.

However, UML diagrams are more practical and user-friendly at the design level than flattened and expanded automata. For instance, UML 2 [9] proposes various notations to express in a concise and efficient manner time and duration constraints in addition to concurrency and communication aspects (see Figure 14). Particularly, to overcome shortcomings of UML 1.x. the new version of UML introduces a new sub-package SimpleTime which adds meta-classes to represent time and durations as well as actions to observe time passing. However, this package fits for time annotations on sequence diagrams rather than on state diagrams, mainly in terms of expressing time constraints between different occurrences of events. One thus can use them for describing protocols of ports whose interactions are time constrained (along with protocol state machines for the other ports). Moreover, since UML sequence diagrams can also be translated into timed automata using formalization methods (e.g., [7]) they can be used for conformance checking similarly to protocol state machine. The only information added is the time constraints on nodes and arcs of timed automata.

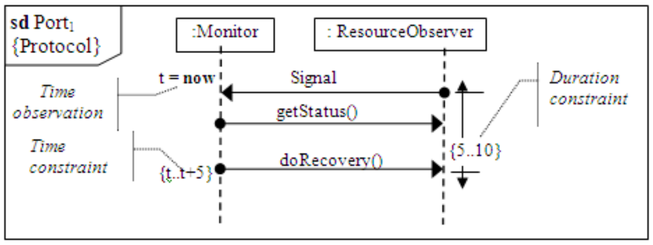


Figure 14. Sequence Diagram with Timing Constraints of Port$_1$.

5. Conclusion

This paper presented a formal method to check the conformance of interaction protocols of components with their behavior models given in terms of UML state machines. The notion of allowed operations on protocol transitions are extended for active components to encompass operations and signals, each of which may be trigged either by the environment and the component itself since it possesses its owns thread of control and can thus progress according to its own pace. Both of behavior and protocol artifacts are translated into more abstract models in order to compare them and check their properties using analysis tools.

Besides protocol state machines, sequence diagrams are investigated as a modeling language of interaction protocols, especially time constrained ones.

For future work, we plan to find out the conditions to extend the results of conformance checking of components to that of the whole system. We plan as well, to combine various formal techniques such abstract interpretation and theorem proving along with model checking to alleviate the verification task reactive systems and particularly, overcome the state explosion problem.

References

[1] R. Alur and D. Dill, “A Theory of timed automata”, Theoretical Computer Science, Volume 126, pp. 183-235, Elsevier, 1994.
[2] K. Bierhoff, M. Kehrt, S. Han, D. Saini, M. Al-Meshari, and J. Aldrich, “A Language-based Approach to Specification and Enforcement of Architectural Protocols”, Technical Report CMU-ISR-10-110, School of Computer Science, Carnegie Mellon University, March 2010.
[3] Handbook of Process Algebra, edited By J.A. Bergstra, A. Ponse, and S.A. Smolka, 2001, ISBN 0444828303, Elsevier Science Inc., NewYork, USA.
[4] A. Both, and W. Zimmermann, “Automatic protocol conformance checking of recursive and parallel component-based systems”, In Proceedings of 11th International Symposium on Component-Based Software Engineering, (CBSE’2008), LNCS 5282, pp.163-179. Springer, 2008.
[5] K Havelund, M Lowry, S Park, C Pecheur, J Penix, W Visser, and JL White, “Formal Analysis of the Remote Agent Before and After Flight”, The 5th NASA Langley Formal Methods Workshop, Virginia, June 2000.
[6] Y. Hammal, “A formal Semantics of UML State Charts by means of Timed Petri Nets”, In Proceedings of the 25th International Conference on Formal Techniques for Networked and Distributed Systems, LNCS 3731, Springer, Taiwan, 2-5 Oct. 2005.
[7] Y. Hammal, “Branching Time Semantics for UML 2.0 Sequence Diagrams”, In Proceedings of the 26th International Conference on Formal Techniques for Networked and Distributed Systems, LNCS 4229, Springer, Paris, 26-29 Sept. 2006.
[8] Magee J., and Kramer J., “Concurrency -State Models and Java Programs”, John Wiley & Sons, March 1999, 353 pages(Wiley -Worldwide Series in Computer Science).
[9] Object Management Group, Inc. (OMG), “Unified Modeling Language: Superstructure version 2.3”, May 2010.Available from http:// www.omg.org.
[10] T. B. Trinh, A. H. Truong, and V. H. Nguyen, “Checking Protocol-Conformance in Component Models using Aspect Oriented Programming”, In Proceedings of Advances in Computer Science and Engineering, 2009.
[11] Kim G. Larsen, Paul Pettersson and Wang Yi, “Uppaal in a Nutshell”, In Springer International Journal of Software Tools for Technology Transfer 1(1+2), 1997.