Modular Description and Verification of Concurrent Objects

Jean Paul Bahsoun[1], Stephan Merz[2], Corinne Servières[1]

[1] Institut de recherche en informatique de Toulouse, Toulouse, France
[2] Institut für Informatik, Universität München, Munich, Germany

Abstract. The design of large and complex distributed systems requires a modular approach to support reuse and verification. We propose an object-oriented programming model based on concurrently executing communicating agents (concurrent objects) and an associated proof methodology that exploits the class hierarchy to allow for modular verification.
We propose to separate protocol from functionality in class definitions, and advocate separate hierarchies of protocol classes as a way to overcome the inheritance anomaly of concurrent object-oriented programming.
We formalize an agent in Lamport's Temporal Logic of Actions. Modular verification is achieved by restricting inheritance in a way that ensures that subclasses refine superclasses. Interesting properties can thus be verified at an abstract level, ignoring unnecessary implementation detail.

1 Introduction

The object-oriented paradigm in software construction promises modular, incremental design and has therefore become very attractive to practitioners. On the other hand, its semantical foundations are still not well-understood, and few formalisms [3, 10] have been defined that allow the verification of object-oriented programs. In particular, it is notoriously hard to exploit the modular structure of object (or class) definitions in reasoning about object-oriented systems.

The lack of theoretical underpinnings is even more acute for languages that combine concurrency and object-oriented concepts, although this combination appears intuitively compelling. One of the complications that arise with concurrency is that enabling conditions have to be associated with the methods offered by an object, but that, in a parallel setting, client objects have no way of ensuring that a particular method is currently enabled. We call the collection of synchronization code that ensures that methods are executed only if they are currently enabled the *protocol* of an object. There are well-known examples where seemingly minor changes to the method suite of an object require big, non-local revisions of its protocol. This problem has been called the *inheritance anomaly* of concurrent objects in [15]; it invalidates the idea of constructing objects incrementally by localized updates to existing code.

This paper has two objectives: On the one hand, we suggest to alleviate the inheritance anomaly by distinguishing protocol code from code that provides the functionality of an object. A method can be associated with any number of protocols, which are defined in separate protocol classes, with a class hierarchy of their own. On the other hand, we propose a step towards a compositional proof system for concurrent objects, using Lamport's Temporal Logic of Actions [13] (TLA), which has proven to be an expressive and flexible formalism to specify and verify reactive systems.

We try to avoid being overly formal, but mainly proceed by a few running examples, using an illustrative syntax to present the basic ideas. Also, we do not discuss all aspects of our programming model or proof system, but concentrate on problems connected with inheritance.

Section 2 introduces our programming model, presents agents and agent classes, inheritance, and protocol classes, and shows why this separation contributes to solving the inheritance anomaly. Section 3 then shows how we formalize concurrent objects in TLA in a modular way, and how this is useful in modular reasoning about objects and object-based systems. Finally, section 4 discusses related approaches, limitations of our present model, and future work.

2 A model of concurrent objects

In this section we introduce our object-oriented programming model, where several concurrent objects (which we also call agents) may work concurrently. The agents interact with each other by message passing; we consider several modes of communication such as asynchronous and synchronous messages. We use a class-based notation, where agents are defined as instances of classes. New classes may be constructed by inheritance from previously defined classes, to allow for organization and reuse. We also argue to separate functionality and protocol of agents, and introduce separate protocol classes.

2.1 Agents, actions, and messages

The basis of the system is an agent, which acts like a sequential process and may execute concurrently with other agents. With every agent we associate:

- a unique identity,
- a private state, represented by variables (also called attributes) that are visible only to the agent itself. A variable can store data or (the identity of) another agent; this allows to model delegation and embedded objects. In particular, every agent may refer to its own identity via the predefined (constant) attribute *myself*. We distinguish between control variables and structure variables, as we explain below.
- a set of actions that constitute the agent's interface, subdivided into two disjoint subsets of *methods* and *reflexes*. Methods are executed in response to requests from other agents, while reflexes are executed spontaneously by the agent. Reflexes occur mostly at the specification level, they can be used to model actions performed by the environment or system actions triggered by events that are not visible at the current level of abstraction.

In any given state, some actions defined for the agent may be disabled. For example, a buffer can only output an object if it is not empty. We therefore associate every action with its guard, which must hold true before the action is executed. A request to execute an action that is currently disabled is delayed until the guard becomes true.

Syntactically, we express the body of an action by multiple (and possibly conditional) assignments, much as in UNITY [8], and primitives to send messages.

We insist on separating an agent's attributes into two disjoint sets of *control* and *structure* variables. The guard of a method may only contain control variables. Similarly, we subdivide the assignment section of a method into updates to the control variables and updates to the structure variables. We call *synchronization constraint* or *protocol* the guard of a method plus the assignments that update its control variables. Although this separation may appear unnecessary and complex for very small toy classes, we will see that it helps to overcome problems linked to defining classes by inheritance. Moreover, the domains of control variables are usually finite-state, which helps with the automated analysis of protocols, although we do not consider this issue in this paper.

Agents interact with each other by means of messages. We represent a message in transit by a quintuple $(s, d, m, t, \mathbf{x})$ where s is the identity of the sender, d is the identity of the destination agent, m is the mode of communication (explained below), t is the type of the message which should be the name of a method defined for the receiver, and finally, $\mathbf{x}$ is the parameter list of the message. We consider three modes of communication between agents:

– synchronous communication: after sending a message, the sender waits for a reply or acknowledgement from the receiver before carrying on with its task,
– asynchronous communication: after sending a message, the sender doesn't wait for a receiver reply or acknowledgement,
– semi-synchronous communication: after sending a message, the sender may continue with its task unless it wants to transmit another message of the same type to the same receiver, in which case it will be blocked until receipt of an acknowledgement of the first message from the receiver. Semi-synchronous message passing ensures that the order of messages of a given type between two agents is preserved and may therefore be used to establish a FIFO channel.

For the interconnection network that links the agents of a system, we assume that every message sent will eventually arrive at its destination. Because of the distributed nature of our system model, messages may get out of order, and there may be arbitrary delays. Every agent maintains a queue of unprocessed messages. Whenever a synchronous or semi-synchronous message arrives at an agent, an acknowledgement is sent back to the sender of the message.

2.2 Agent classes

We use classes as templates to describe agents. This section explains base classes that give complete descriptions of agents. Section 2.4 explains how classes may

```
class simple_buffer
  structure                   -- structure variables
    value : integer
  control                     -- control variables
    empty : boolean := true
  method put(x:integer)
    when empty                -- guard of put
    =>   empty := false       -- update of control variables
    do   value := x           -- update of structure variables
  method get(obj:agent)
    when not empty
    =>   empty := true
    do   send_asy(obj, "reply", value)
end simple_buffer.
```

Fig. 1. A class of simple buffers

inherit from existing ones.

Figure 1 contains the definition of a class of simple buffers that may store one integer value and offers two methods: *put* stores a value in the buffer if the buffer is empty, *get* sends the contents of the buffer to another object, using a message of type *reply*, and empties the buffer.

In general, the description of a class of agents consists of:

- the class name,
- possibly initialization parameters supplied at the time of agent creation (we will see examples later),
- the attributes, divided into control and structure variables, and
- the list of action definitions. As explained above, we distinguish between methods and reflexes. Agents of class simple_buf offer the methods *put* and *get*, but do not have reflexes. An action definition consists of
  - the name of the action,
  - its guard, indicated by the keyword **when**,
  - updates to the agent's control variables, preceded by =>, and
  - the proper method body which contains updates to the agent's structure variables and message transmissions. This section is indicated by the keyword **do**.

Conceptually, an agent executes an infinite loop, in which it chooses and executes some enabled action: A reflex is enabled if its guard evaluates to true. A method is enabled if its guard evaluates to true and a request to execute the method has been received from the environment. The choice between several enabled actions is arbitrary, but the implementation must be fair. Thus, we consider individual agents to behave as sequential processes, which allows us to model executions of actions as being atomic from the point of view of the environment.

```
class distributed_buffer(max: integer)
  structure
    mbuf : array [0..max − 1] of simple_buffer
    bottom : 0..(max − 1) := 0                -- index for enq
    top : 0..(max − 1) := 0                   -- index for deq
  method enq(x:integer)
    do   send_ssy(mbuf[bottom], "put", x),
         bottom := (bottom + 1) mod max
  method deq(obj:agent)
    do   send_ssy(mbuf[top], "get", obj),
         top := (top + 1) mod max
end distributed_buffer.
```

Fig. 2. A distributed buffer

2.3 Agent systems

A software system is composed of several agents that work in parallel and communicate by means of messages. In this paper, we ignore the issue of dynamic creation of agents, but focus on static agent networks. Let us give an example of a system of agents.

We will use the class simple_buffer to define an agent which implements a distributed circular buffer that offers the methods *enq* and *deq* to enqueue and dequeue a value. This distributed buffer, whose definition appears in figure 2, embeds an array of simple_buffer agents by storing their identities in its private state. The distributed_buffer agent organizes the simple_buffer agents (which still work in parallel) in a circular fashion. The encapsulation concept allows for a hierarchy of agents, and enables to close the encapsulated agent with respect to any communication with the environment of the encapsulating agent. So our definition of the *distributed_buffer* precludes any communication between the *simple_buffer* agents and any clients of the *distributed_buffer*.

When a distributed_buffer agent receives an *enq* message (resp. a *deq* message), it sends a *put* message (resp. a *get* message) to a simple_buffer agent, using a circular policy. The distributed_buffer is an unbounded buffer because we don't have guards to accept an *enq* or a *deq* request, and the queues storing waiting requests are unbounded. But, of course, requests for *put* or *get* methods can be delayed at the level of simple_buffer agents. The circular policy is ensured thanks to:

- the use of two indexes *bottom* for *enq* and *top* for *deq*, and
- the use of semi-synchronous communication.

To illustrate some other features of our concept, figure 3 contains the definition of a very simple client-server system. An agent of class server manages the access to some resource. When executing method *Enter1*, the server sends a message *Receive1* to the client that requested the resource, informing it that the resource is free, and ready to be used.

```
class server1
  control
    in1: boolean := false
  method Enter1(c : client1)
    when not in1
    => in1 := true
    do  send_asy(c, "Receive1")
  method Leave1
    when in1
    => in1 := false
end server1.
```

```
class client1(s : server1)
  control
    owner1, waiting1 : boolean := false
  reflex Request1
    when (not owner1) and (not waiting1)
    => waiting1 := true
    do  send_asy(s, "Enter1", myself)
  method Receive1
    => (owner1, waiting1) := (true, false)
  reflex Return1
    when owner1
    => owner1 := false
    do  send_asy(s, "Leave1")
  reflex Use1
    when owner1
    do  -- use resource
end client1.
```

Fig. 3. Class definitions for a simple client-server system

Class client1 is parameterized by the server1 agent the client will be using. The client contains three reflexes *Request1*, *Return1*, and *Use1* to request access to the resource, relinquish the resource, or use it. In an implementation, some of these reflexes could be methods called by the client itself or some other agent.

2.4 Inheritance

We can define a class from existing ones by inheritance. The new class (called the subclass) inherits existing attributes and actions, but it may define new attributes, offer new actions, and update or redefine inherited actions.

Suppose we want to extend the behaviour of the agents of the server class: now they will have to synchronize the access to two resources that can not be used simultaneously. Figure 4 shows the server2 class, which extends the server1 class. Similarly, we define another class client2 of clients by inheritance from client1, which may use either resource controlled by the server. A server of class server2 can be used by a client of either class client1 or client2.

The class server2 inherits the control variable *in1* and the methods *Enter1* and *Leave1*. In this subclass, we add a control variable *in2* and two methods *Enter2* and *Leave2* to control access to the second resource. We also have to update the synchronization constraint of method *Enter1* to forbid the access to the first resource when the second one has been allocated. In general, we write updates to actions in the form

when *condition* => *control_upd* **do** *structure_upd*

much as we write action definitions. The guard of the updated method is the conjunction of the inherited guard and the condition in the update clause. The

```
class server2 inherit server1
  control
    in2: boolean := false
  method Enter2(c : client2)
    when (not in1) and (not in2)
    => in2 := true
    do  send_asy(c, "Receive2")
  method Leave2
    when in2
    => in2 := false
  update method Enter1
    when not in2
end server2.
```

```
class client2(s : server2) inherit client1
  control
    owner2, waiting2 : boolean := false
  reflex Request2
    when (not owner2) and (not waiting2)
    => waiting2 := true
    do  send_asy(s, "Enter2", myself)
  method Receive2
    => (owner2, waiting2) := (true, false)
  reflex Return2
    when owner2
    => owner2 := false
    do  send_asy(s, "Leave2")
  reflex Use2
    when owner2
    do  -- use resource 2
end client2.
```

Fig. 4. Another client-server system

effect of the updated action is to simultaneously perform the inherited assignments and message transmissions as well as those newly defined in *control_upd* and *structure_upd*. (As before, we do not allow multiple assignments to the same variable.)

Alternatively, we may also completely redefine actions in the subclass. In this case, we give the entire definition of the action, using the keyword **redefine** instead of **update**.

On the other hand, we will see that we can obtain pleasant proof rules by using updates and redefinitions in subclasses in a restricted way. Formally, we say that a subclass *preserves encapsulation* of its superclass if the following conditions are met:

- New methods do not modify inherited attributes.
- Updates to inherited actions may only strengthen the guards.

Note that both server2 and client2 preserve encapsulation of their respective superclasses.

2.5 Protocol classes

We observe that in the two classes simple_buffer and server, the synchronization constraints of the methods have the same shape. In fact, many software systems with different functionality share the same protocol. It would therefore be useful to have a mechanism to reuse synchronization constraints in several classes. Such a mechanism, based on the syntatic separation between the definitions of synchronization constraints and the methods that use them, has already been suggested in [5, 4]. A class can thus be built as a composition of a class of

```
protocol class mutex
  control
    inUse: boolean := false
  protocol acquire
    when not inUse
    =>   inUse := true
  protocol release
    when inUse
    =>   inUse := false
end mutex.
```

```
class server1 constrained by mutex
  method Enter1(c : client1)
    use protocol acquire
    do    send_asy(c, "Receive1")
  method Leave1
    use protocol release
    do -- nothing
end server1.
```

Fig. 5. Definition of server1 using a protocol class

methods that implement the functionality and a protocol class that defines the desired synchronization constraints.

A protocol class may declare control variables like an ordinary class. Instead of actions, it declares named protocols that consist of a guard and updates to the control variables declared in the protocol class. A protocol class can be inherited by an ordinary class, using the keyword **constrained by** instead of **inherits**. In this case, the control part of the protocol class is added to the control part of the new class, and any protocol defined in the protocol class can be referenced by its name in the new class. More precisely, the inherited constraints can be combined with the constraints of the new class as explained in section 2.4 above.

Let us illustrate these ideas using our examples. First, we define a protocol class called mutex, that will be reused twice to define other classes. We can build up the class server1 using the mutex protocol class as illustrated in figure 5. The server class defined in this way is identical (up to the name of the local control variable) to the server class given in figure 3.

The keyword **use protocol** can appear in the declaration of an action before its guard (if there is any), and it is followed by a list of names of inherited protocols. These protocols are combined with the constraints explicitly defined in the action declaration and those inherited from superclasses, to yield the effective synchronization constraint.

As another example, figure 6 defines the class cell: The method *put* allows to store a new value to the cell, erasing any previous value stored in the cell; the current value can be read using method *get*, even if the cell hasn't been initialized.

As shown in figure 6, class simple_buffer can be defined by inheritance from cell and mutex; this definition is again equivalent to that shown in figure 1.

2.6 The inheritance anomaly

The term "inheritance anomaly" has been coined in [15] to designate the problem of having to update or redefine all of the inherited method definitions when introducing some new method that interferes with the inherited protocol. In this section, we will illustrate at the hand of three classical examples from [15]

```
class cell
  structure
    value : integer
  method put(x:integer)
    do  value := x
  method get(obj:agent)
    do  send_asy(obj, "reply", value)
end simple_buffer.
```

```
class simple_buffer inherit cell
constrained by mutex
  update method put
    use protocol acquire
  update method get
    use protocol release
end simple_buffer.
```

Fig. 6. Another definition of simple_buffer

```
class distributed_buffer2(max: integer)
inherits distributed_buffer
  method deq2(obj:agent)
    do    send_ssy(mbuf[top], "get", obj),
          send_ssy(mbuf[(top + 1) mod max], "get", obj),
          top := (top + 2) mod max
end distributed_buffer2.
```

Fig. 7. Extending class distributed buffer with method *deq2*

that the separation of functionality and protocol alleviates the problem, because we can simply refine the protocol. All of the examples are based on the distributed_buffer class defined in figure 2.

First suppose that we want to define a new method *deq2* which removes the two values at the top of the buffer. Observe that the effect of this method will in general be different from a client sending two consecutive *deq* messages to the distributed_buffer agent because of interference from messages from other agents. However, using semi-synchronous communication between the distributed_buffer and its embedded simple_buffer agents, as shown in figure 7, achieves the desired functionality.

Second, we add a method *ddeq* to class distributed_buffer to obtain another subclass distributed_buffer_ddeq. Method *ddeq* behaves like *deq*, except that it must not be performed immediately after executing an *enq* method. Such protocol patterns occur quite frequently: some actions of a class may be sensitive to the execution of a "disabling" action. The protocol class disable of figure 8 captures this pattern of behaviour: the control variable *disabled* is true iff the last action performed was a "disabling" action. For our particular, example, the definition of class distributed_buffer_ddeq is defined by inheritance from class distributed_buffer and disable, giving the definition of method *ddeq*, and updating the synchronization constraint of the remaining actions.

The third example from [15] is conceptually similar: We now want to define two new methods *open* and *close*. The idea is that after executing a *close* method, all methods except *open* are disabled, and can be performed only after an explicit request to reopen the buffer. Figure 9 shows the definition of an appropriate protocol class lock as well as the definition of a class lockable_distributed_buffer based on distributed_buffer and lock. In particular, lockable_distributed_buffer preserves

```
protocol class disable
  control
    disabled : boolean := false
  protocol sensitive
    when not disabled
  protocol insensitive
    when true
    => disabled := false
  protocol disable :
    when true
    => disabled := true
end disable.
```

```
class distributed_buffer_ddeq
inherit distributed_buffer
constrained by disable
  method ddeq(obj: agent)
  use protocol sensitive
    do  send_ssy(mbuf[top], "get", obj),
        top := (top + 1) mod max
  update method deq
    use protocol insensitive
  update method enq
    use protocol disable
end distributed_buffer_ddeq.
```

Fig. 8. Adding method *ddeq* to class distributed_buffer

```
protocol class lock
  control
    locked : boolean := false
  protocol lockable
    when not locked
  protocol close
    when not locked
    => locked := true
  protocol open :
    when locked
    => locked := false
end disable.
```

```
class lockable_distributed_buffer
inherit distributed_buffer
constrained by lock
  method open
  use protocol open
  method close
  use protocol close
  update method deq
    use protocol lockable
  update method enq
    use protocol lockable
end lockable_distributed_buffer.
```

Fig. 9. Adding *open* and *close* methods to the buffer

encapsulation of both its superclasses.

3 Formalizing agent behaviour in TLA

We will now discuss verification of agents and agent systems in a modular way. As a basis, we give an axiomatic semantics of agents using Lamport's Temporal Logic of Actions (TLA) [13]. We chose TLA because a state-based formalism seems to correspond well to the concept of state inherent in the object-oriented approach to programming. Moreover, the basic concepts of TLA have been carefully chosen such that many proof obligations can be deferred to nontemporal reasoning, enabling the use of existing theorem provers [7, 12]. Since TLA formulas, in contrast to most temporal logics, are invariant under stuttering, we get simple formal notions of refinement and parallel composition [1].

In our formalization, we propose to distinguish between two levels of reasoning about systems built from concurrently executing agents, each focusing on one particular aspect or view of agent systems.

At the *agent level*, we reason about behaviors of individual agents. We take advantage of the encapsulation of an agent's private state, which may only change by executing methods or reflexes defined for the agent. We think of an agent as an open system, and will describe aspects of communication and interaction with other agents in the form of environment assumptions. We will show that properties of superclasses can be reused in reasoning about subclasses as long as the subclass preserves encapsulation of the superclass.

The *system level* models the top-level view of the entire system by an external observer. System properties should be deduced from properties of single agents and assumptions about the communication network, in a compositional way; this form of composition is orthogonal to modular reasoning about class hierarchies.

Since TLA does not have a built-in notion of "locality" and does not enforce the proof methodology outlined above, it may in fact turn out that it is not the most suitable logic for our programming model, and we are studying more adapted logics [18]. However, the basic ideas are independent of the particular logic.

3.1 Describing a base agent

We will first explain how we model an agent of a base class in TLA. With a declaration of class A we associate a formula (we write multi-line conjunctions and disjunctions as lists bulleted by $\wedge$ or $\vee$)

$$\begin{aligned}\Phi_{\mathsf{A}}(a) \;\triangleq\; & \wedge\; \mathit{Init}_{\mathsf{A}}(a) \;\wedge\; \Box[\mathcal{N}_{\mathsf{A}}(a)]_{\mathit{attr}_{\mathsf{A}}(a)} \\ & \wedge \bigwedge_{m \in \mathit{Meth}_{\mathsf{A}}} \Box[(\exists x : \mathit{receive}(a, m, x)) \;\vee\; m(a)]_{a.q_m} \\ & \wedge\; \mathit{Fairness}_{\mathsf{A}}(a)\end{aligned}$$

to formalize the behaviour of an agent a of class A. We model the agent a as a record that contains one field for every attribute of class A (we let $\mathit{attr}_{\mathsf{A}}(a)$ denote the tuple of these fields). Besides, the record contains a field q_m for every message type $m \in \mathit{Meth}_{\mathsf{A}}$ defined in class A, representing the queue of waiting requests to execute method m. The tuple of these fields is denoted by $\mathit{Msg}_{\mathsf{A}}(a)$.

The predicate $\mathit{Init}_{\mathsf{A}}(a)$ defines the initial conditions for the components of record a. The initial conditions for the attributes can be directly read off the variable initializations in the declaration of class A. Additionally, we require all mail queues to be initially empty (we use ε to denote the empty sequence).

The action $\mathcal{N}_{\mathsf{A}}(a)$ represents the next-state relation for the attributes of a; it is defined as a disjunction

$$\mathcal{N}_{\mathsf{A}}(a) \;\triangleq\; \bigvee_{\alpha \in \mathit{Act}_{\mathsf{A}}} \alpha(a)$$

of the TLA formulas representing the actions defined for class A. These formulas are straightforward to read off the class declaration, using a predefined action $\mathit{send}_a(\mathit{ms})$ [6] that describes the emission of a message set ms by agent a. Additionally, we add a conjunct for every method declared in class A that states that the first message waiting in the associated mail queue is consumed.

$$
\begin{aligned}
\mathit{Init}_{\mathsf{server1}}(a) &\triangleq \neg a.\mathit{in1} \;\wedge\; a.q_{\mathit{Enter1}} = \varepsilon \;\wedge\; a.q_{\mathit{Leave1}} = \varepsilon \\
\mathit{Enter1}(a,c) &\triangleq \wedge\; a.q_{\mathit{Enter1}} = \langle c \rangle \circ a.q'_{\mathit{Enter1}} \\
&\;\; \wedge\; \neg a.\mathit{in1} \;\wedge\; a.\mathit{in1}' \;\wedge\; \mathit{send}_a(\{\langle c, \mathsf{asy}, \text{“Receive1”}, \langle\rangle\rangle\}) \\
\mathit{Leave1}(a) &\triangleq \wedge\; a.q_{\mathit{Leave1}} = \langle\rangle \circ a.q'_{\mathit{Leave1}} \\
&\;\; \wedge\; a.\mathit{in1} \;\wedge\; \neg a.\mathrm{in1}' \;\wedge\; \mathit{send}_a(\emptyset) \\
\mathcal{N}_{\mathsf{server1}}(a) &\triangleq (\exists c : \mathit{Enter1}(a,c)) \;\vee\; \mathit{Leave1}(a) \\
\mathit{Fairness}_{\mathsf{server1}}(a) &\triangleq (\forall c : \mathrm{WF}_{a.\mathit{in1}}(\mathit{Enter1}(a,c))) \;\wedge\; \mathrm{WF}_{a.\mathit{in1}}(\mathit{Leave1}(a)) \\
\Phi_{\mathsf{server1}}(a) &\triangleq \wedge\; \mathit{Init}_{\mathsf{server1}}(a) \;\wedge\; \Box[\mathcal{N}_{\mathsf{server1}}(a)]_{a.\mathit{in1}} \\
&\;\; \wedge\; \Box[\exists c : \mathit{receive}(a, \mathit{Enter1}, \langle c \rangle) \;\vee\; \mathit{Enter1}(a,c)]_{a.q_{\mathit{Enter1}}} \\
&\;\; \wedge\; \Box[\mathit{receive}(a, \mathit{Leave1}, \langle\rangle) \;\vee\; \mathit{Leave1}(a)]_{a.q_{\mathit{Leave1}}} \\
&\;\; \wedge\; \mathit{Fairness}_{\mathsf{server1}}(a)
\end{aligned}
$$

Fig. 10. TLA formalization of class server1

The action $\mathit{receive}(a, m, \mathbf{x})$ describes the receipt of a message of type m by agent a with parameter tuple $\mathbf{x}$ (of appropriate length); it is defined as

$$\mathit{receive}(a, m, \mathbf{x}) \;\triangleq\; a.q'_m = a.q_m \circ \langle \mathbf{x} \rangle$$

Consequently, the conjunct $\Box[(\exists x : \mathit{receive}(a, m, x) \;\vee\; m(a)]_{a.q_m}$ states that the mail queue $a.q_m$ associated with method m for agent a only changes during the receipt of a new message of this type, or during the execution of method m by agent a.

Finally, the formula $\mathit{Fairness}_{\mathsf{A}}(a)$ describes the fairness conditions associated with class A. These fairness conditions have been left implicit in the programming notation, but will be made explicit in the logical description. For the purposes of this paper, we simply assume weak fairness for all actions defined for an agent.

As a complete example, figure 10 contains the TLA representation of agents of class server1 defined in figure 3.

3.2 Describing an agent defined by inheritance

An agent defined by inheritance can be formalized in the same way as an agent of a base class. We will, however, here restrict ourselves to subclasses that preserve encapsulation of their superclasses, because we can formalize and reason about agents of such classes in a compositional manner.

A subclass B of class A may define additional attributes and actions as well as update existing actions by strenghtening their guards. We will in the following denote the sets of additional attributes, actions, and methods declared in B by $\mathit{attr}_{\mathsf{B}}$, $\mathit{Act}_{\mathsf{B}}$, and $\mathit{Meth}_{\mathsf{B}}$, and the set of actions from A that are updated in B by $\mathit{Upd}_{\mathsf{B}}$. The new guard associated with an updated action $\alpha \in \mathit{Upd}_{\mathsf{B}}$ for agent b is denoted by $\mathcal{G}_{\mathsf{B},\alpha}(b)$.

The general form of the TLA formalization of an agent b of subclass B of class A will then be

$$\begin{aligned}\Phi_{\mathsf{B}}(b) \;\triangleq\; \Phi_{\mathsf{A}}(b) \wedge\ & \mathit{Init}_{\mathsf{B}}(b) \;\wedge\; \Box[\mathcal{N}_{\mathsf{B}}(b) \wedge \mathit{attr}_{\mathsf{A}}(b)' = \mathit{attr}_{\mathsf{A}}(b)]_{\mathit{attr}_{\mathsf{B}}(b)} \\ \wedge\ & \bigwedge_{m \in \mathit{Meth}_{\mathsf{B}}} \Box[(\exists x : \mathit{receive}(c, m, x)) \;\vee\; m(b)]_{b.q_m} \\ \wedge\ & \Box[\mathit{attr}_{\mathsf{B}}(b)' = \mathit{attr}_{\mathsf{B}}(b) \wedge \bigwedge_{\alpha \in \mathit{Upd}_{\mathsf{B}}} \alpha(b) \;\Rightarrow\; \mathcal{G}_{\mathsf{B},\alpha}(b)]_{\mathit{attr}_{\mathsf{A}}(b)} \\ \wedge\ & \mathit{Fairness}_{\mathsf{B}}(b)\end{aligned}$$

The first conjunct states that agent b behaves at least like an agent of the superclass. This ensures that b may be used whenever an agent of class A is expected, which is essential for a robust class hierarchy. The remaining conjuncts assert the additional properties that b has to satisfy.

The initial conditions for the new attributes can be read off the definition of class B as before. Moreover, we require all mail queues corresponding to the new methods $m \in \mathit{Meth}_{\mathsf{B}}$ to be empty.

Similarly, the action $\mathcal{N}_{\mathsf{B}}(b)$, which models the new actions defined for class B, is defined as in section 3.1 above. The next-state relation for the attributes in $\mathit{attr}_{\mathsf{B}}$ requires the inherited attributes to stay unchanged; this corresponds to our assumption that class B preserves encapsulation of class A. The next conjunct asserts that the mail queues for the additional methods from $\mathit{Meth}_{\mathsf{B}}$ behave as expected.

The following conjunct asserts that whenever the inherited attributes of b change, the new attributes stay unchanged (because inherited actions do not modify new attributes) and that the new, stronger guard has to hold whenever an updated action is executed.

Finally, formula $\mathit{Fairness}_{\mathsf{B}}(b)$ adds fairness conditions for the newly defined actions as in section 3.1 above.

For an agent b of class server2, which has been defined by inheritance from class server1 in figure 4, we obtain the TLA formalization shown in figure 11.

3.3 Reasoning about an agent defined by inheritance

We will now indicate how the formal description of an agent by a TLA formula helps us to reason about local properties of the agent. More precisely, we want to obtain modular proof rules that reflect the definition of the agent by inheritance, provided the subclass preserves encapsulation of the superclass.

First, the formula that describes the superclass appears as a conjunct in the characterization of the subclass. Hence, all properties of the superclass are obviously inherited. To prove additional properties of the subclass, we will make use of the structure of its TLA description.

Suppose we want to verify that an agent of the class server2 guarantees that at no time both of the resources it manages are in use:

$$\Phi_{\mathsf{server2}}(b) \;\Rightarrow\; \Box\neg(b.\mathit{in1} \;\wedge\; b.\mathit{in2})$$

$$
\begin{aligned}
\mathit{Init}_{\mathsf{server2}}(b) &\triangleq \neg b.\mathit{in2} \;\wedge\; b.q_{\mathit{Enter2}} = \varepsilon \;\wedge\; b.q_{\mathit{Leave2}} = \varepsilon \\
\mathit{Enter2}(b,c) &\triangleq \wedge\; b.q_{\mathit{Enter2}} = \langle c\rangle \circ b.q'_{\mathit{Enter2}} \\
&\;\; \wedge\; \neg b.\mathit{in2} \;\wedge\; b.\mathit{in2}' \;\wedge\; \mathit{send}_b(\{\langle c, \mathsf{asy}, \text{“Receive2”}, \langle\rangle\rangle\}) \\
\mathit{Leave2}(b) &\triangleq \wedge\; b.q_{\mathit{Leave2}} = \langle\rangle \circ b.q'_{\mathit{Leave2}} \\
&\;\; \wedge\; b.\mathit{in2} \;\wedge\; \neg b.\mathrm{in2}' \;\wedge\; \mathit{send}_b(\emptyset) \\
\mathcal{N}_{\mathsf{server2}}(b) &\triangleq (\exists c : \mathit{Enter2}(b)) \;\vee\; \mathit{Leave2}(b) \\
\mathit{Fairness}_{\mathsf{server2}}(b) &\triangleq (\forall c : \mathrm{WF}_{b.\mathit{in2}}(\mathit{Enter2}(b,c))) \;\wedge\; \mathrm{WF}_{b.\mathit{in2}}(\mathit{Leave2}(b)) \\
\Phi_{\mathsf{server2}}(b) &\triangleq \Phi_{\mathsf{server1}}(b) \;\wedge\; \mathit{Init}_{\mathsf{server2}}(b) \;\wedge\; \Box[\mathcal{N}_{\mathsf{server2}}(b) \wedge b.\mathit{in1}' = b.\mathit{in1}]_{b.\mathit{in2}} \\
&\qquad\qquad\qquad \wedge\; \Box[\exists c : \mathit{receive}(b, \mathit{Enter2}, \langle c\rangle) \;\vee\; \mathit{Enter2}(b,c)]_{b.q_{\mathit{Enter2}}} \\
&\qquad\qquad\qquad \wedge\; \Box[\mathit{receive}(b, \mathit{Leave2}, \langle\rangle) \;\vee\; \mathit{Leave2}(b)]_{b.q_{\mathit{Leave2}}} \\
&\qquad\qquad\qquad \wedge\; \Box[b.\mathit{in2}' = b.\mathit{in2} \wedge \forall c : \mathit{Enter1}(b,c) \;\Rightarrow\; \neg b.\mathit{in2}]_{b.\mathit{in1}} \\
&\qquad\qquad\qquad \wedge\; \mathit{Fairness}_{\mathsf{server2}}(b)
\end{aligned}
$$

Fig. 11. TLA formalization of class server2

The usual approach to proving invariants [13] is to prove that they are true initially, and that every action preserves the invariant. Formally, we are left with the proof obligations

$$
\begin{aligned}
&(1)\; \mathit{Init}_{\mathsf{server1}}(b) \;\wedge\; \mathit{Init}_{\mathsf{server2}}(b) \;\Rightarrow\; \neg(b.\mathit{in1} \;\wedge\; b.\mathit{in2}) \\
&(2)\; \wedge\; [\mathcal{N}_{\mathsf{server1}}(b)]_{b.\mathit{in1}} \\
&\quad\;\; \wedge\; [\mathcal{N}_{\mathsf{server2}}(b) \wedge b.\mathit{in1}' = b.\mathit{in1}]_{b.\mathit{in2}} \\
&\quad\;\; \wedge\; [b.\mathit{in2}' = b.\mathit{in2} \wedge \forall c : \mathit{Enter1}(b,c) \;\Rightarrow\; \neg b.\mathit{in2}]_{b.\mathit{in1}} \\
&\quad\;\; \Rightarrow\; (\neg(b.\mathit{in1} \;\wedge\; b.\mathit{in2}) \;\Rightarrow\; \neg(b.\mathit{in1}' \;\wedge\; b.\mathit{in2}'))
\end{aligned}
$$

Proving (1) is easy, given the definitions from figures 10 and 11. Formula (2) looks more intimidating. In fact, we would like to avoid to construct the complete next-state relation of the attributes of agent b from the definitions of $\mathcal{N}_{\mathsf{server1}}(b)$ and $\mathcal{N}_{\mathsf{server2}}(b)$, but rather reason separately about the next-state relations of the subcomponents. This is mostly a matter of convenience for small examples like this one, but it becomes a necessity if we want to deal with realistic examples without running into state explosion problems. Fortunately, modular reasoning is quite simple, thanks to the structure of our specifications. Formally, the following proposition allows to reason about subclasses that preserve encapsulation.

Proposition 1 *Let class* B *be a subclass of class* A *that preserves encapsulation, and let* $\Phi_{\mathsf{B}}(b)$ *be the TLA formula that describes an agent of class* B *as defined in section 3.2. Let* p *be any action formula of TLA.*

If (i) $\mathcal{N}_{\mathsf{B}} \;\wedge\; \mathit{attr}_{\mathsf{A}}(b)' = \mathit{attr}_{\mathsf{A}}(b) \;\Rightarrow\; p$ *and*

(ii) $\mathcal{N}_{\mathsf{A}} \;\wedge\; \mathit{attr}_{\mathsf{B}}(b)' = \mathit{attr}_{\mathsf{B}}(b) \;\wedge\; \bigwedge_{\alpha \in \mathit{Upd}_{\mathsf{B}}} \alpha(b) \;\Rightarrow\; \mathcal{G}_{\mathsf{B},\alpha}(b) \;\Rightarrow\; p$

then $\Phi_{\mathsf{B}}(b) \;\Rightarrow\; \Box[p]_{\langle \mathit{attr}_{\mathsf{A}}(b), \mathit{attr}_{\mathsf{B}}(b)\rangle}$ *holds.*

For our example of class server2, we choose

$$
p \triangleq \neg(b.\mathit{in1} \;\wedge\; b.\mathit{in2}) \;\Rightarrow\; \neg(b.\mathit{in1}' \;\wedge\; b.\mathit{in2}')
$$

which leaves us with proving

$$\mathcal{N}_{\mathsf{server2}} \;\wedge\; b.\mathit{in1}' = b.\mathit{in1} \;\;\Rightarrow\;\; p$$
$$\mathcal{N}_{\mathsf{server1}} \;\wedge\; b.\mathit{in2}' = b.\mathit{in2} \;\wedge\; (\forall c : \mathit{Enter1}(b, c) \;\Rightarrow\; \neg b.\mathit{in2}) \;\Rightarrow\; p$$

Both of these proofs are straightforward from the definitions of figures 10 and 11.

3.4 Reasoning about agent systems

We formalize an agent system as a parallel composition of its individual agents and a specification of the communication network. We have shown in sections 3.1 and 3.2 how we can formalize single agents by TLA formulas. A generic specification of a communication network for agent systems has been proposed in [6], and we will not give details in this paper.

Let us assume that we have a simple network of one agent s of class $\mathsf{server1}$ and two clients $c1$ and $c2$ of class $\mathsf{client1}(s)$. This network is then formally described by the conjunction

$$\Phi_{\mathsf{server1}}(s) \;\wedge\; \Phi_{\mathsf{client1}(s)}(c1) \;\wedge\; \Phi_{\mathsf{client1}(s)}(c2) \;\wedge\; \Phi_{\mathsf{Net}}$$

of formulas that represent the individual agents and the network.

We deduce system properties from agent properties and properties of the network. As an example, let us prove that every request of the client $c1$ to obtain the resource from the server s will eventually be honoured:

$$\mathit{Request1}(c1) \leadsto \Diamond \mathit{Receive1}(c1) \qquad (*)$$

(The "leadsto" formula $F \leadsto G$ abbreviates $\Box(F \;\Rightarrow\; \Diamond G)$.)

We deduce this property from the following local properties of the single agents and the network:

(1) $(\forall c : \mathit{Enter1}(c) \leadsto \mathit{Leave1}) \;\Rightarrow\; (\mathit{receive}(s, \mathit{Enter1}, \langle c \rangle) \leadsto \mathit{Enter1}(s, c))$
(2.1) $\mathit{receive}(c1, \mathit{Receive1}, \langle\rangle) \leadsto \mathit{Receive1}(c1)$
(2.2) $\mathit{receive}(c2, \mathit{Receive1}, \langle\rangle) \leadsto \mathit{Receive1}(c2)$
(3) $\mathit{receive}(s, \mathit{Leave1}, \langle\rangle) \leadsto \mathit{Leave1}(s)$
(4.1) $\mathit{Receive1}(c1) \leadsto \mathit{Return1}(c1)$
(4.2) $\mathit{Receive1}(c2) \leadsto \mathit{Return1}(c2)$
(5) $\mathit{send}_{c1}(\{\langle s, \mathsf{asy}, \mathit{Enter1}, \langle c1 \rangle\rangle\}) \leadsto \mathit{receive}(s, \mathit{Enter1}, \langle c1 \rangle)$
(6.1) $\mathit{send}_{c1}(\{\langle s, \mathsf{asy}, \mathit{Leave1}, \langle\rangle\rangle\}) \leadsto \mathit{receive}(s, \mathit{Leave1}, \langle\rangle)$
(6.2) $\mathit{send}_{c2}(\{\langle s, \mathsf{asy}, \mathit{Leave1}, \langle\rangle\rangle\}) \leadsto \mathit{receive}(s, \mathit{Leave1}, \langle\rangle)$
(7.1) $\mathit{send}_{s}(\{\langle c1, \mathsf{asy}, \mathit{Receive1}, \langle\rangle\rangle\}) \leadsto \mathit{receive}(c1, \mathit{Receive1}, \langle\rangle)$
(7.2) $\mathit{send}_{s}(\{\langle c2, \mathsf{asy}, \mathit{Receive1}, \langle\rangle\rangle\}) \leadsto \mathit{receive}(c2, \mathit{Receive1}, \langle\rangle)$

Properties (2.1), (2.2), and (3) are local liveness properties that assert that messages received by the agents will eventually be processed. These properties are verified by standard TLA reasoning about the agent specifications. Similarly, properties (4.1) and (4.2) are local liveness properties that assert that the clients will eventually release the resource after obtaining it. They follow from the fairness conditions placed on reflex *Return1*.

Properties (5), (6.1), (6.2), (7.1), and (7.2) are liveness conditions on the network that state that sent messages will eventually be delivered.

Property (1) is more interesting: it states that the server will eventually honour all requests to execute method *Enter1*, assuming that every *Enter1* action is followed by a *Leave1* action. Informally, this requires all agents to eventually give up their resources. Clearly, if some agent keeps the resource forever, the server could not allocate it to another agent that requests access. We will frequently make similar assumptions about the environment behaviour and use them in reasoning about an agent.

We obtain $(*)$ from (1)–(7.2) by a simple chain of reasoning: First, properties (7.1), (2.1), (4.1), (6.1), and (3), and the action definitions imply

$$\mathit{Enter1}(c1) \leadsto \mathit{Leave1}$$

and similarly for agent $c2$. Using (1), we can deduce

$$\mathit{receive}(s, \mathit{Enter1}, \langle c \rangle) \leadsto \mathit{Enter1}(s, c)$$

and $(*)$ follows using properties (5), (7.1), and (2.1).

Using a similar combination of local reasoning and facts about the communication network, we may conclude that the clients $c1$ and $c2$ access the resource in mutual exclusion, that is,

$$\Box\neg(c1.\mathit{owner1} \ \wedge \ c2.\mathit{owner1}$$

4 Conclusion

The purpose of this paper has been to emphasize the importance of modularity in both program construction and verification. We have indicated a programming model based on concurrently executing sequential agents that communicate by message passing. This model appears both flexible and semantically tractable: it can readily be formalized using existing program logics such as TLA.

We advocate the separation of protocols and functionality as a way to overcome the inheritance anomaly [15] of concurrent objects. We have therefore introduced two parallel hierarchies of classes that provide functionality and protocol classes that express synchronization constraints. More generally, we may think of an object as being composed of small, relatively independent components that can be understood—and verified—separately. The inheritance anomaly has recently received a great deal of attention [11, 17, 14]. Examining the successful solutions, it appears that all rely on modular description of synchronization code. We believe that our proposal, which can be understood as pushing this idea as far as possible, is particularly simple and robust.

Compositionality is an essential prerequisite to avoid the state explosion problem in verification. It allows proof outlines to be designed such that basic verification tasks are small enough to be verified by machine and the proof structure can be checked by interactive reasoning tools.

Object-oriented concurrent systems provide two orthogonal modes of composition: First, agent systems are composed from agents that execute in parallel. This form of composition has been studied in the theory of parallel processes for many years and is rather well understood [9]. In TLA, parallel composition simply corresponds to conjunction.

Another form of composition corresponds to subclassing. It is clear that it is in general impossible to inherit properties from the parent classes, unless some discipline on modification is imposed. We have therefore suggested that subclasses preserve encapsulation of the superclasses they inherit from. We may then obtain well-structured descriptions of the subclasses, and preserve all properties derived for the parent classes. New properties can be deduced in a modular way that avoids constructing the entire next-state relation. On the other hand, the condition of preserving encapsulation is probably too restrictive for many practical applications, and it could be relaxed in a number of ways. For example, we could allow the subclass to modify inherited attributes as long as these changes can be expressed in terms of inherited actions.

Other approaches to the formal definition of concurrent objects include, most notably, the Actor model [2], the language Maude [16], as well as the POOL family of languages [3]. With the exception of relatively minor differences such as assumptions on the order of processing of incoming messages and the detailed semantics of message-passing primitives, the basic programming models of these approaches are quite similar to ours. However, few formalisms come with a logic to reason about behavioural properties of agents. For example, all reasoning in the Actor framework has to be carried out at the semantic level. The rules of rewrite logic underlying Maude, on the other hand, do not allow to prove correctness assertions, but only support properties that express the existence of a successful execution.

References

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 81(2):253–284, May 1991.
2. G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*. MIT Press, Cambridge, Mass., 1987. [19, pages 37–53].
3. Pierre America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 161–168, Berlin, 1990. Springer-Verlag.
4. Jean Paul Bahsoun and Louis Féraud. A model to design reusable parallel software components. In D. Etiemble and J.-C. Syre, editors, *Parallel Architectures and Languages Europe (PARLE)*, number 630 in Lecture Notes in Computer Science, pages 245–260, Berlin, 1992. Springer-Verlag.
5. Jean Paul Bahsoun, Louis Féraud, and C. Bétourné. A two degrees of freedom approach for parallel programming. In *International Conference on Computer Languages*, pages 261–270. IEEE, 1990.

6. Jean Paul Bahsoun, Stephan Merz, and Corinne Servières. A framework for programming and formalizing concurrent objects. In *SIGSOFT Conference 1993*. IEEE, December 1993.
7. Holger Busch. A practical method for reasoning about distributed systems in a theorem prover. In E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors, *Workshop on Higher-Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 106–121, Berlin, 1995. Springer-Verlag.
8. K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison-Wesley, Reading, Mass., 1988.
9. W. P. de Roever. The quest for compositionality – a survey of proof systems for concurrency, part 1. In E. J. Neuhold, editor, *IFIP working group on the role of abstract models in Computer Science*, pages 181–206. North-Holland, 1985.
10. José Luiz Fiadeiro and Thomas Maibaum. Verifying for reuse: foundations of object-oriented system verification. In C.Hankin, I.Makie, and R.Nagarajan, editors, *Theory and Formal Methods 1994*. World Scientific Publishing Company, 1995.
11. S. Frølund. Inheritance of synchronisation constraints in concurrent object-oriented programming languages. In O. Lehrmann Madsen, editor, *European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 185–196, Berlin, 1992. Springer-Verlag.
12. Thomas Långbacka. A HOL formalisation of the Temporal Logic of Actions. In Thomas E Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 332–345, Berlin, 1994. Springer-Verlag.
13. Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
14. Ulrike Lechner, Christian Lengauer, Friederike Nickl, and Martin Wirsing. (OO + concurrency) & reusability—how to overcome the inheritance anomaly. Technical report, Institut für Informatik, Universität München, Munich, 1996.
15. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, Cambridge, Mass., 1993.
16. J. Meseguer. Parallel programming in Maude. In J. B. Banatre and D. LeMetayer, editors, *Research Directions in High-Level Parallel Programming Languages*, Lecture Notes in Computer Science. Springer-Verlag, 1992.
17. José Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In Oscar Nierstrasz, editor, *European Conference on Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 220–246, Berlin, 1993. Springer-Verlag.
18. Corinne Servières. *Modélisation et vérification orientées objet pour les systèmes réactifs*. PhD thesis, Institut national polytechnique de Toulouse, Toulouse, November 1995.
19. A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming*. MIT Press, Cambridge, Mass., 1987.

This article was processed using the LaTeX macro package with LLNCS style