Modelling a Multi-Agent System Relating to Liveness Properties in Event-B

Lorina NEGREANU
University POLITEHNICA of Bucharest,
313 Splaiul Independentei, Bucharest, 060042, Romania,
lorina.negreanu@cs.pub.ro

Abstract: Safety and liveness are properties of a formal model that ensure the correct and continuous progress of the model. The aim of this paper is to present a formal modelling and proof of correctness for a multi-agent system for requesting services, with respect to liveness properties - fairness and starvation freedom. The model is specified and verified using the Event-B formalism and the Rodin platform - an Eclipse plug-in meant to allow the writing and checking specification correctness. Event-B is a formal method based on first-order logic and set theory as an underlying mathematical notation used to model and reason about complex and discrete systems. One central mechanism of Event-B modelling is the concept of refinement that allows building a model in a step by step fashion, by adding more details to an initially abstract model, in order to reduce the level of abstraction, thus making it closer to reality. In our development we used refinement techniques, constructing an ordered sequence of embedded models, where each of them is a refinement of the one preceding it in the sequence.

Keywords: multi-agent systems, formal methods, Event-B, refinement, liveness, validation.

1. Introduction

We strongly believe that the rigorous development of complex systems should be based on mathematical models which can be analyzed by doing proofs. The obvious target is to reduce the number of design faults. The model of a system has a declarative semantics that allows us to prove that the defined properties of the system are consistent and will be present in it.

Multi-agent systems are complex, distributed, reactive systems that are quite difficult to specify formally. Event-B has been used to model multi-agent systems with a focus on concepts such as mobility and trust [5] or autonomy and interaction with a common environment [7]. Our aim is to model some other important properties of such systems.

This paper presents a formal modelling of liveness properties of a multi-agent system for requesting services, that will be integrated in an ambient intelligent system. The model is built using the Event-B [1], [3] formal specification method and machine checked using the Rodin tool [12], [13]. This should be read as an extension of the model we previously specified in [9] and [10]. The paper emphasises on the further refinements of the model (2nd and 3rd refinements) that enable us to check the liveness properties of the multi-agent system.

Safety and liveness are properties of a formal model that ensure the correct and continuous progress of the model at hand. While a safety property specifies that something bad will never happen [6] - "the system never reaches a bad state" - for example, some property holds throughout the execution (deadlock freedom, mutual exclusion), a liveness property specifies that something desirable will eventually happen [6] - "there is progress in the system" - some actions occur infinitely often. Our interest is in modelling liveness properties of a given system, more precisely, fairness and starvation freedom [4]. Fairness properties state that "if something is enabled sufficiently often, then it must eventually happen". We consider the typical fairness assumption (strong fairness) that enforces an event to be taken sufficiently often, but also the property that rather prevents a particular choice from being taken sufficiently often [14]. This becomes important in our model when the requests are satisfied. It is possible that a request cannot be satisfied for an indefinite period of time while other requests continue normally, which may occur if the satisfying scheme of the requests is unfair. On the other hand the availability time of a service may be insufficient and subsequently lead to starvation due to timeouts.

Liveness in an Event-B model is based on deadlock and live-lock freedom.

Since the Event-B language does not provide any facility to state liveness properties [8], we require liveness assumptions over some events to obtain a deadlock and live-lock-free model.
The paper is organized as follows: Section 2 describes the multi-agent system under scrutiny, Section 3 presents the Event-B specification of the initial model as well as further refinements which are done with respect to liveness properties. Section 4 contains the model validation. Section 5 lists the conclusions.

2. System Description

Consider a user who intends to obtain a service in a relaxing center (e.g. swimming, physiotherapy or talking to somebody) that is located in a smart house; the smart house permanently monitors the person in order to detect any changes in condition (for medical and monitoring purposes). Since the person's well-being is important, we need a system to manage his requests based on some measured parameters.

The system we propose is composed of several measured parameters.

- an agent to interrogate the ambient factors - Temperature Agent;
- an agent to verify the health status of the supervised person - Pulse Agent;
- an User Agent associated with the user;
- service Agents (each agent is responsible with a specific service);
- Community Agent (this agent knows all the available/unavailable services).

We consider n Service Agents each having a specialization and a maximum available time. All the Service Agents are connected to the Community Agent. The User Agent assists the user in requesting the desired service.

In order to get access to a service, the user makes a request. The User Agent analyzes the request by computing the priority and duration of the service using the monitoring information (health status of the person and performed activities). The User Agent sends a verification message to the Temperature Agent and to the Pulse Agent in order to verify if the supervised person can perform the activity. For instance, if the supervised person wants to meet someone and it is too hot outside or he has a high pulse, the Temperature Agent or the Pulse Agent measures the temperature/pulse and decides whether or not to cancel the requested service.

If the parameters are in regular bounds, the User Agent sends a request message to the Community Agent (new_request with the priority for the requested service and the requested duration). If the answer for the request is not received in a predefined time, the User Agent sends a message to the Community Agent (modify_request) in order to increase the priority of the requested service for that user. The request is cancelled by the User Agent if the request is still not feasible after a specified deadline; thus a cancel_request message is sent to the Community Agent. If there is enough available time for the requested service, the corresponding Service Agent informs the Community Agent who sends a satisfy_request to the User Agent.

The Service Agents are connected to the Community Agent. Some services may be requested more frequently than others, therefore the Community Agent might request an increase of the maximum available duration for a Service Agent, by taking some available time from other services; a request_available message is sent to each of the Service Agents. The Service Agent associated with a service with fewer requests replies with a release_available message in order to add some extra time to that service.

3. Formal Specification

In our previous work [10] we specified the request as the main modelling element. A request is defined as a member of the set of requests. It has a status (pending or satisfied), an associated service and a duration, that can be accessed through the functions: status ∈ requests → STATUS, where status assigns a status to each request, reference ∈ requests → SERVICES, where reference assigns a service to each request, duration ∈ requests → N*, where duration assigns a duration to each request (with the assumption that if a service is requested, the duration is at least 1).

The status of the request is changed into satisfied, if the duration of the requested service is either less than or equal to the time availability of the service, defined as available ∈ SERVICES → N.
In the modelling [10] we started by considering that all requested references exist in the set of available services and that this set and the set of requests are given in an up-to-date state. We derived by refinement [2] the situation in which we have to take into account new requests, modification of requests, cancelation of requests and update of services time availability.

Figure 1 describes the abstract machine and the context associated, that were subject to our previous work [10].

**Figure 1.** The context and the variables of the abstract machine

The context introduces the abstract set of all possible requests (existing and future) - REQUESTS, the abstract set of services - SERVICES, and the set STATUS = \{pending, satisfied\} describing the status of a request. The existing requests are a subset of REQUESTS.

The invariants of the abstract machine are specified in Figure 2 a), and the initialization of the variables in Figure 2 b).

**Figure 2.** a) The invariants and b) initialization of the abstract machine

In order to specify the events we considered that all requested references are references in available and that the satisfy_request event is enabled when the time availability of the requested service is greater than the duration requested. The events cancel_request, new_request, modify_request, request_available and release_available are modelling the state changes for the variables attached to the services and requests. The satisfy_request event is specified in Figure 3.

**Figure 3.** The specification of the satisfy_request event

The events new_request, cancel_request and modify_request are abstractly modelled, specifying only the fact that a modification is possible. The specification of the events is similar, as in Figure 4 (where only the event new_request is specified).

**Figure 4.** The specification of the new_request event

The events request_available and release_available are also abstractly defined:

**3.1 First Refinement**

We have refined the abstract machine [10] by taking into account new requests, and
cancellation of requests. For the current modelling purpose we kept the events modify_request, request_available and release_available abstract. The events new_request and cancel_request are given in Figure 5 a) and b).

new_request:  
  REFINES  
  ANY request  
  WHERE  
    grd1: r ∈ REQUESTS \ requests  
    grd2: d ∈ N  
    grd3: s ∈ SERVICES  
  THEN  
    act1: requests := requests ∪ {r}  
    act2: status(r) := pending  
    act3: duration(r) := d  
    act4: reference(r) := s  
END  

cancel_request:  
  REFINES  
  ANY request  
  WHERE  
    grd1: r ∈ requests  
    grd2: status(r) = pending  
  THEN  
    act1: requests := requests \ {r}  
    act2: status := {r} ⩤ status  
    act3: duration := {r} ⩤ duration  
    act4: reference := {r} ⩤ reference  
END  

Figure 5. First refinement for new_request and cancel_request events

3.2 Second Refinement

The second and third refinements address the modelling of the system under the fairness and starvation freedom assumptions. The specification presented in [10] captures the notion of flow by a set. In fact, it is possible that a request remains always pending and is never satisfied, because there are always other requests which are processed. The solution we envisaged in [9] is to add a priority to each request that is increased the longer it waits and to satisfy the request with the highest priority. If a request is waiting too long (more than a specific deadline) the request is cancelled. The event new_request gives each new request a priority using a parameter $p$ ($p ∈ N$). The variable priority, priority ∈ requests → N, records the priority of the request and the new condition strengthens the guard of the event satisfy_request.

$$∀ p \cdot (p ∈ requests ∧ status(p) = pending ∧ duration(p) ≤ available(reference(p)) \Rightarrow priority(p) ≤ priority(r)).$$

In order to manage the waiting time of a request we add a time stamp recorded in the variable timer, timer ∈ requests → N. The event new_request gives each new request a time stamp using the variable clock, clock ∈ N, that grows larger as each successive operation is invoked: $timer(r) := clock$.

The event cancel_request is enabled if the difference between the current clock value and the time stamp of the request is larger than the deadline of the request: $clock − timer(r) > deadline(r)$, where the constant deadline, deadline ∈ REQUESTS → N, records the maximum time for a request to be processed.

The event modify_request increases the priority of the request if it took longer than the amount of time given by the constant oldies, oldies ∈ REQUESTS → N: $clock − timer(r) > oldies(r)$.

The context of the second refinement is given in Figure 6 a) and the added variables in Figure 6 b).

<table>
<thead>
<tr>
<th>CONTEXT</th>
<th>Services_c1</th>
<th>EXTENDS</th>
<th>Services_c0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTANTS</td>
<td>deadline</td>
<td>PQ</td>
<td>oldies</td>
</tr>
<tr>
<td>AXIOMS</td>
<td>axm1: deadline ∈ REQUESTS → N</td>
<td>axm2: pq ∈ N</td>
<td>axm3: oldies ∈ REQUESTS → N</td>
</tr>
</tbody>
</table>

The invariants for the variables timer, clock and priority are given in Figure 7 a) and the corresponding initialization in Figure 7 b).

```
inv1: timer ∈ requests → ℕ
inv2: clock ∈ ℕ
inv3: priority ∈ REQUESTS → ℕ
```

```
... act6: timer := ∅
act7: clock := 0
act8: priority ∈ REQUESTS → ℕ
```

### Figure 7. a) The invariants and b) initialization for the second refinement

The satisfy_request event is given in Figure 8.

```
satisfy_request:
  REFINES
  satisfy_request
  ANY
  r
  WHERE
  grd1: r ∈ requests
  grd2: status(r) = pending
  grd3: duration(r) ≤ available(reference(r))
  grd4: ∀p·(p ∈ requests ∧ status(p) = pending ∧ duration(p) ≤ available(reference(p)) ⇒ priority(p) < priority(r))
THEN
  act1: status(r) := satisfied
  act2: available(reference(r)) := available(reference(r)) − duration(r)
  act3: clock := clock + 1
END
```

### Figure 8. The satisfy_request event for the second refinement

The events new_request and cancel_request are given in Figure 9 a) and b).

```
new_request:
  REFINES
    new_request
  ANY
  r d s p
  WHERE
  grd1: r ∈ REQUESTS \ requests
  grd2: d ∈ ℕ
  grd3: s ∈ SERVICES
  grd4: p ∈ ℕ
THEN
  act1: requests := requestsU(r)
  act2: status(r) := pending
  act3: duration(r) := d
  act4: reference(r) := s
  act5: priority(r) := p
  act6: timer(r) := clock
  act7: clock := clock + 1
END
```

### Figure 9. The new_request and cancel_request events for the second refinement

The modify_request event is given in Figure 10. The events request_available and release_available are not further refined. They have the same specification.
3.3 Third Refinement

Another problem is that the time availability of the service might not be sufficient for a given request but is infinitely often sufficient for some other requests - the well known starvation problem. An increase of the time availability of a service might allow some other request to be satisfied. Therefore we consider the increase/decrease of the time availability of a service based on the number of requests. The variable \( \text{extra} \) records the extra time from which an amount of time can be requested or released by the services. We define the constant \( \text{timeout} \), \( \text{timeout} \in \text{SERVICES} \rightarrow \mathbb{N} \), that specifies the amount of time that a service can wait for a new request. The variable \( \text{counter} \), \( \text{counter} \in \text{SERVICES} \rightarrow \mathbb{N} \), records the time stamp of the last request for a service. If the difference between the current time (stored in the \( \text{clock} \) variable) and the variable \( \text{counter} \) is larger than the timeout for a service an amount of time might be released. In order to allow a service to request an extra time amount we have to sum the durations of the requests for that service. The variable \( \text{sum} \), \( \text{sum} \in \text{SERVICES} \rightarrow \mathbb{N} \), stores the sum of the requested durations for a service.

The event \( \text{new_request} \) is refined to record the time stamp for a service \( \text{counter}(r) := \text{clock} \), and to add the requested duration to the sum: \( \text{sum}(s) := \text{sum}(s) + d \). When a request is cancelled, the duration has to be removed from the sum: \( \text{sum}(\text{reference}(r)) := \text{sum}(\text{reference}(r)) - \text{duration}(r) \).

The event \( \text{cancel_request} \) is refined to reflect the change. When the sum of the requested durations for a service becomes larger than the time availability of that service: \( \text{available}(s) < \text{sum}(s) \), and there is enough extra time: \( dq \leq \text{extra} \), the event \( \text{request_available} \) might be enabled, and the time availability of the service is increased by a time amount, recorded in the constant \( dq \), \( dq \in \mathbb{N} \): \( \text{available}(s) := \text{available}(s) + dq \). On the other hand if there are no requests coming in the timeout interval for a service \( s \): \( \text{clock} - \text{counter}(s) > \text{timeout}(s) \), and the time availability is large enough: \( dq \leq \text{available}(s) \) then the event \( \text{release_available} \) might be enabled and the time availability of the service is decreased by the time quantum \( dq \): \( \text{available}(s) := \text{available}(s) - dq \). Also the service gets a new time stamp.

The context and the variables for the third refinement are given in Figure 11 a) and b).

The invariants and the specific initialization are given in Figure 12 a) and b).

\[
\begin{align*}
\text{modify_request} &: \quad \text{REFINES} \\
& \quad \text{modify_request} \\
\text{ANY} & \quad r \\
\text{WHERE} & \quad \text{grd1: } r \in \text{requests} \\
& \quad \text{grd2: status}(r) = \text{pending} \\
& \quad \text{grd3: clock} - \text{timer}(r) > \text{oldies}(r) \\
\text{THEN} & \quad \text{act1: priority}(r) := \text{priority}(r) + pq \\
& \quad \text{act2: clock} := \text{clock} + 1 \\
\end{align*}
\]

Figure 10. The \textit{modify_request} event for the second refinement

\[
\begin{align*}
\text{CONTEXT} & \quad \text{Services_c2} \\
\text{EXTENDS} & \quad \text{Services_c1} \\
\text{CONSTANTS} & \quad \text{timeout} \\
& \quad \text{dq} \\
\text{AXIOMS} & \quad \text{axm1: } \text{timeout} \in \text{SERVICES} \rightarrow \mathbb{N} \\
& \quad \text{axm2: } \text{dq} \in \mathbb{N} \\
\end{align*}
\]

\[
\begin{align*}
\text{MACHINE} & \quad \text{Services}_3 \\
\text{REFINES} & \quad \text{Services}_2 \\
\text{SEES} & \quad \text{Services}_c2 \\
\text{VARIABLES} & \quad \text{extra} \\
& \quad \text{sum} \\
& \quad \text{counter} \\
\end{align*}
\]

Figure 11. The context and the variables for the third refinement

\[
\begin{align*}
\text{inv1: } & \quad \text{extra} \in \mathbb{N} \\
\text{inv2: } & \quad \text{sum} \in \text{SERVICES} \rightarrow \mathbb{N} \\
\text{inv3: } & \quad \text{counter} \in \text{SERVICES} \rightarrow \mathbb{N} \\
\text{inv4: } & \quad \text{priority} \in \text{REQUESTS} \rightarrow \mathbb{N} \\
\text{act9: } & \quad \text{extra} := 0 \\
\text{act10: } & \quad \text{sum} := \text{SERVICES} \rightarrow \mathbb{N} \\
\text{act11: } & \quad \text{counter} := \text{SERVICES} \rightarrow \mathbb{N} \\
\end{align*}
\]

Figure 12. a) The invariants and b) initialization for the third refinement
The \texttt{satisfy\_request} event is not further refined. The definition of the \texttt{cancel\_request} event is given in Figure 13.

\begin{verbatim}
cancel_request:  
  REFINES cancel_request  
  ANY r  
  WHERE  
    grd1: r ∈ requests  
    grd2: status(r) = pending  
    grd3: clock − timer(r) > oldies(r)  
  THEN  
    act1: sum(reference(r)) := sum(reference(r)) − duration(r)  
    act2: requests := requests\{r\}  
    act3: status := \{r\} \triangleleft status  
    act4: duration\{r\} \triangleleft duration  
    act5: reference\{r\} \triangleleft reference  
    act6: priority\{r\} \triangleleft priority  
    act7: timer := \{r\} \triangleleft timer  
    act8: clock := clock + 1  
END
\end{verbatim}

Figure 13. The \texttt{cancel\_request} event

The \texttt{new\_request} and \texttt{modify\_request} events are given in Figure 14 a) and b).

\begin{verbatim}
new_request:  
  REFINES new_request  
  ANY r d s p  
  WHERE  
    grd1: r ∈ REQUESTS \ requests  
    grd2: d ∈ \mathbb{N}\{1\}  
    grd3: s ∈ SERVICES  
    grd4: p ∈ \mathbb{N}  
  THEN  
    act1: requests := requests \{r\}  
    act2: status(r) := pending  
    act3: duration(r) := d  
    act4: reference(r) := s  
    act5: priority(r) := p  
    act6: timer(r) := clock  
    act7: counter(s) := clock  
    act8: sum(s) := sum(s) + d  
    act9: clock := clock + 1  
END
\end{verbatim}

\begin{verbatim}
modify_request:  
  REFINES modify_request  
  ANY r  
  WHERE  
    grd1: r ∈ requests  
    grd2: status(r) = pending  
    grd3: clock − timer(r) > oldies(r)  
  THEN  
    act1: priority(r) := priority(r) + pq  
    act2: clock := clock + 1  
END
\end{verbatim}

Figure 14. The \texttt{new\_request} and \texttt{modify\_request} events

The events \texttt{request\_available} and \texttt{release\_available} are given in Figure 15 a) and b).

\begin{verbatim}
request_available:  
  REFINES request_available  
  ANY s  
  WHERE  
    grd1: s ∈ SERVICES  
    grd2: available(s) < sum(s)  
    grd3: dq ≤ extra  
  THEN  
    act1: available(s) := available(s) + dq  
    act2: extra := extra − dq  
    act3: clock := clock + 1  
END
\end{verbatim}

\begin{verbatim}
release_available:  
  REFINES release_available  
  ANY s  
  WHERE  
    grd1: s ∈ SERVICES  
    grd2: clock − counter(s) > timeout(s)  
    grd3: dq ≤ available(s)  
  THEN  
    act1: extra := extra + dq  
    act2: available(s) := available(s) − dq  
    act3: counter(s) := clock  
    act4: clock := clock + 1  
END
\end{verbatim}

Figure 15. The \texttt{request\_available} and \texttt{release\_available} events
4. Model Validation

Most of the proofs are done automatically by Rodin, but there are certain cases where human interaction is necessary to complete the task of proving. The proof statistics show that 78 proof obligations were generated by the Rodin platform [12], [13]. 75 proof obligations were discharged automatically while the others were discharged by interactive proofs.

The initial model as well as the further refinements has been checked for deadlock freeness using ProB [11]. All models have been successfully checked.

5. Conclusions

In this paper, we have presented a specification of a multi-agent system for requesting services with respect to liveness properties. We proceeded by constructing a series of models, where the initial model specifies the system requirements and the final one describes the resulting system. We used the Rodin tool for Event-B to prove that each successive model refines the previous one, whereby the resulting system is correct by construction.

Our aim was to provide a specification focused on liveness properties - fairness and starvation freedom. We proved these properties by appropriately combining invariants, event refinement, and deadlock freedom. As multi-agent systems are reactive systems our formalization should be useful for specifying different properties of other kinds of reactive systems.

REFERENCES