https://doi.org/10.1007/s00165-021-00567-y

Formal Aspects of Computing

Formal Aspects of Computing

A refinement-based development of a distributed signalling system

Paulius Stankaitis[1], Alexei Iliasov[4], Tsutomu Kobayashi[2], Yamine Aït-Ameur[3], Fuyuki Ishikawa[2], Alexander Romanovsky[1]

[1] School of Computing, Newcastle University, Newcastle upon Tyne, United Kingdom
[2] National Institute of Informatics, Tokyo, Japan
[3] INPT–ENSEEIHT, 2 Rue Charles Camichel, Toulouse, France
[4] The Formal Route Limited, 32A Woodhouse Grove, E12 6SR, London, United Kingdom

Abstract. The decentralised railway signalling systems have a potential to increase capacity, availability and reduce maintenance costs of railway networks. However, given the safety-critical nature of railway signalling and the complexity of novel distributed signalling solutions, their safety should be guaranteed by using thorough system validation methods. To achieve such a high-level of safety assurance of these complex signalling systems, scenario-based testing methods are far from being sufficient despite that they are still widely used in the industry. Formal verification is an alternative approach which provides a rigorous approach to verifying complex systems and has been successfully used in the railway domain. Despite the successes, little work has been done in applying formal methods for distributed railway systems. In our research we are working towards a multifaceted formal development methodology of complex railway signalling systems. The methodology is based on the Event-B modelling language which provides an expressive modelling language, a stepwise development and a proof-based model verification. In this paper, we present the application of the methodology for the development and verification of a distributed protocol for reservation of railway sections. The main challenge of this work is developing a distributed protocol which ensures safety and liveness of the distributed railway system when message delays are allowed in the model.

Keywords: Distributed railway signalling, Distributed resource allocation, Event-B method

1. Introduction

Railway signalling is a safety-critical system whose responsibility is to guarantee a safe and efficient operation of railway networks. In the last few decades there have been proposals to utilise distributed system concepts in the railway signalling as a way of increasing railway network capacity and reducing maintenance costs (e.g. [HP00, WK12]). These emerging distributed railway signalling concepts propose to use a radio-based communication technology and novel distributed signalling protocols for decentralising contemporaneous signalling systems.

Correspondence to: Paulius Stankaitis, e-mail: paulius.stankaitis@newcastle.ac.uk

Published online: 24 November 2021

Because of their complex concurrent behaviour, distributed systems are notoriously difficult to validate and this could curtail the development, and deployment of novel distributed signalling solutions. To achieve a high-level of safety assurance of novel distributed signalling systems popular railway system validation methods, such as scenario-based testing methods, would be far from being sufficient. In recent years, there has been a push by industries with a strong focus on distributed systems to incorporate formal methods into their system development processes [HHK+17, New14]. The railway domain has proved to be a fruitful area for applying various formal methods, but considerably less has been done in applying them for distributed railway systems by industry and academia [BBFM99, ED06]. Therefore, the long-term aim of our research is to lower the barriers of applying formal methods for the development of complex railway signalling systems, including distributed [SIK+20], heterogeneous [SI17] and hybrid [SDS+19] railway systems.

In order to manage the modelling and verification complexity of complex railway systems we are working towards a multifaceted formal development methodology. The methodology primarily relies on three concepts: a refinement-based model development, communication modelling patterns [SIA+19] and a proof-based model verification. The methodology is built upon the Event-B method [Abr13] which provides an expressive modelling language, flexible refinement mechanism and proof-based verification. It is also paramount that the methodology should support quantitative evaluation; as stated by Fantechi and Haxthausen [FH18], signalling solutions will only be adopted in practice if system availability is demonstrated. Therefore, in our proposed multifaceted methodology we also integrate a stochastic simulator for a quantitative railway system analysis.

In this paper, we present our work on a refinement-based formal development of a novel distributed signalling protocol. The paper attempts to demonstrate the advantages of the proposed methodology and refinement-based formal development, particularly, in the early stages of the system development (system prototyping) where specifications and requirements are incomplete. The main objective of the developed protocol was to guarantee a safe allocation of railway sections while ensuring liveness of the distributed signalling system. In our work, the liveness property of the system had to be guaranteed by the protocol in a model where message delays are permitted. The authors of related researches did not consider liveness and fairness properties which, as discussed in [FH18], can directly affect system availability. The consideration of message delays has also introduced modelling and deductive verification challenges as subtle deadlock scenarios were discovered.

Related work. In [FH18] the authors formalise the railway interlocking problem as a distributed mutual exclusion problem and discuss the related literature on distributed interlocking (e.g. [HP00, FHN17, WK12]). In principle all railway models share similar high-level safety, liveness and fairness requirements, as summarised on page 2 in [FH18]. One difference between our work and the studies overviewed in [FH18] is the interlocking engineering concept and the system model (e.g. allowed message delays). Another difference is the formal consideration of liveness and fairness requirements. In our work we not only prove the safety properties of the protocol, but also ensure systems liveness, fairness and analyse performance. A similar distributed signalling concept is presented as a case study in [Pro16]. The authors verified their system design via a simulation approach and only considered scenarios with up to two trains. In our verification approach we prove the distributed signalling system mathematically and hence guarantee its safety for any number of trains. In the study by Morley [Mor96] the author formally proved a distributed protocol, which is used in the real-world railway signalling systems to reserve a route, which is jointly controlled by adjacent signalling systems. Even though the distributed signalling concepts of our works are different, the effects of message delays to the safety were considered in both works. Our developed distributed signalling protocol is also based on serialisability and similar to ones defined from transactions processing [EGLT76, BSR80, GR92] in centralised and distributed database systems.

Paper structure. The rest of the paper is organised in the following way. Section 2 revisits a previously proposed multifaceted methodology and provides background information on the Event-B formal specification language. In Sect. 3 we elicit high-level distributed system requirements and specifications as well as provide a semi-formal description of the distributed resource allocation protocol. In Sect. 4 we present the formal protocol model development with the Event-B method. In the last section, Sect. 5, we summarise our work and discuss future work directions.

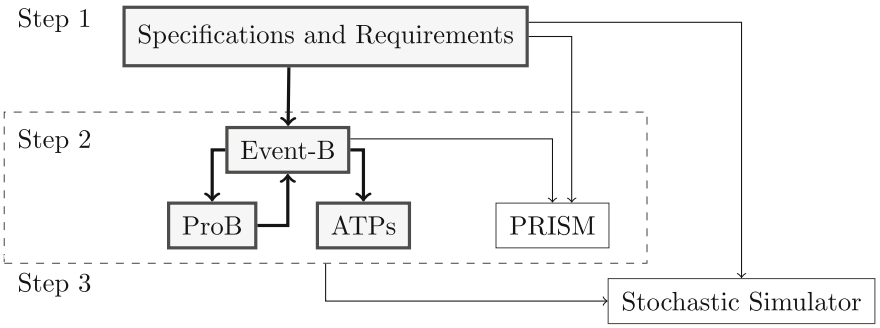

Fig. 1. Multifaceted modelling and verification framework (with focus of the paper highlighted)

event_label $\widehat{=}$ $[specification_{id}]$
ANY
parameters
WHEN
guards
THEN
actions

INVARIANTS
invariant_label invariant $[requirement_{id}]$

Fig. 2. An example of annotated Event-B model event and invariant in the Rodin platform

2. Multifaceted formal development methodology

The generally accepted approach to any formal system development is starting a development process by first informally describing system specifications and requirements. Therefore, this is the first step (Step 1 in Fig. 1) in our formal development process. At the moment we do not suggest or provide a systematic approach to defining requirements and specifications of a distributed system, however, we require labelling each requirement with a unique label. By labelling requirement and specifications we intend to ensure the traceability between informal and formal methodology artefacts. The traceability aspect guarantees the completeness of the model, meaning, that all informal specifications and requirements have been captured by the formal model. On the formal Event-B artefact, we require to annotate events and invariants of the Event-B model with labelled references to a specific informal artefact (see Fig. 2).

The following step (Step 2) in our methodology is a development and verification of a functional formal Event-B model. The purpose of formally modelling a distributed system is to have a formal artefact, which can be animated, analysed and formally verified. For the development and verification of functional system models we selected the Event-B [Abr13] specification language, which has been successfully used for a formal development of various distributed protocols (e.g. [CM06, HKBA09, ILTR11]). The Event-B method provides an expressive modelling language, flexible refinement mechanism and is also proof-driven, meaning that model correctness is demonstrated by generating and discharging proof obligations with available automated theorem provers [DFGV14, ISAYR16]. The method is also supported by tools such as ProB [LB03] which makes it possible to animate a model and validate it via model-checking. ProB can be particularly useful in early modelling stages where it could be too onerous to deductively verify a model. On the other hand, the Event-B method does not have an adequate probabilistic reasoning support, which was essential to verifying our distributed railway section reservation protocol. Therefore, it was decided to integrate a well-known stochastic model checker, PRISM [HKNP06], into the framework, so the probabilistic properties of a system can be verified.

The last step (Step 3) in the proposed formal development process is analysing performance of the developed distributed system. For analysing system's performance we have implemented a high-fidelity protocol simulator which could help to evaluate a protocol under normal or *stressed* conditions. Because of the paper focus on Steps 1 and 2, in the following sections we provide more detail on the functional Event-B model development and the Event-B method.

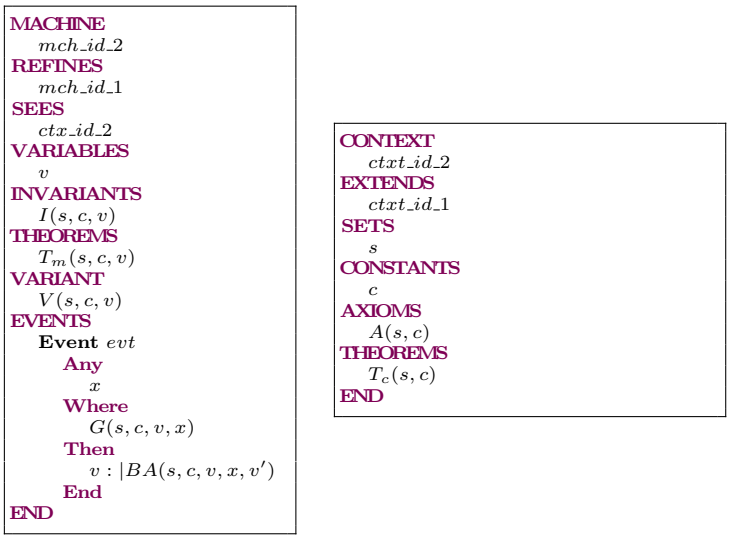

Fig. 3. Event-B model structure

Table 1. Event-B proof obligations

Theorems	$A(s, c) \Rightarrow T_c(s, c)$
Invariant Preservation	$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x) \wedge BA(s, c, v, x, v') \Rightarrow I(s, c, v')$
Event Feasibility	$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x) \Rightarrow \exists v' \cdot BA(s, c, v, x, v')$
Variant Progress	$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x) \wedge BA(s, c, v, x, v') \Rightarrow V(s, c, v') < V(s, c, v)$

2.1. Event-B

The Event-B mathematical language [Abr13] used in the system development and analysis is an evolution of the classical B method [Abr96] and Action Systems [Bac90]. The formal specification language offers a fairly high-level mathematical language based on a first-order logic and Zermelo-Fraenkel set theory as well as an economical yet expressive modelling notation. The formalism belongs to a family of state-based modelling languages where a state of a discrete system is simply a collection of variables and constants whereas the transition is a guarded variable transformation.

A cornerstone of the Event-B method is the step-wise development that facilitates a gradual design of a system implementation through a number of correctness preserving refinement steps. The model development starts with a creation of a very abstract specification and the model is completed when all requirements and specifications are covered. The Event-B model is made of two key components-machines and contexts which respectively describe dynamic and static parts of the system (see Fig. 3). The context contains modeller declared constants c and associated axioms $A(s, c)$ which can be made visible in machines. The dynamic part of the model contains variables v which are constrained by invariants $I(s, c, v)$ and initialised by an action. The state variables are then transformed by actions which are part of events and the modeler may use predicate guards $G(s, c, v, x)$ to denote when event is triggered.

The Event-B method is a proof driven specification language where model correctness is demonstrated by generating and discharging proof obligations-theorems in the first-order logic. Table 1 shows a few of more important proof obligations of the Event-B language. The model is considered to be correct when all proof obligations are discharged.

Rodin [The06] is an open source Eclipse-based integrated development environment (IDE) for Event-B model development. The Rodin is a core set of plug-ins for project management, formal development, syntactic analysis, proof assistance and proof-based verification.

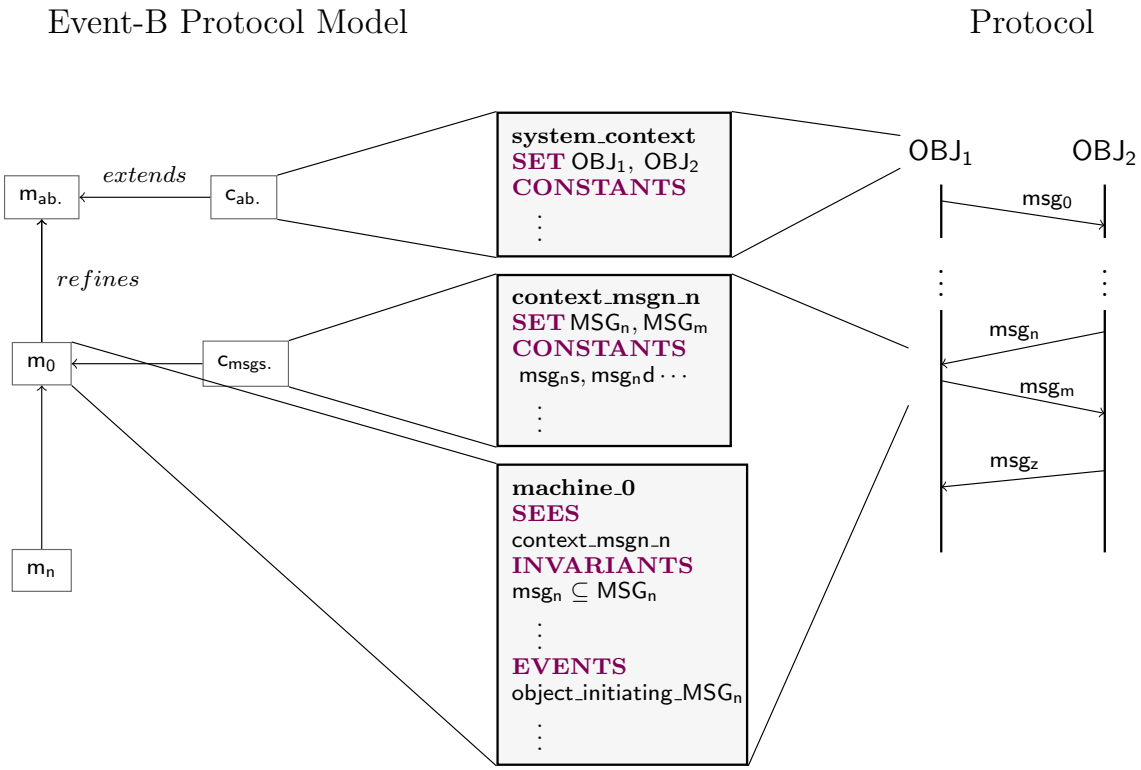

Fig. 4. Protocol modelling patterns

2.2. Developing functional protocol models in Event-B

A formal functional Event-B model can have a multitude of uses, but the main application is for formally proving properties about the distributed system. The completed distributed system's model in Step 2 should cover all requirements and specifications, and would be considered correct when all generated proof obligations are proved.

The model development approach we utilise (visualised in Fig. 4) is a rather standard and starts with the abstract model $m_{ab.}$ which formally specifies the objective of the distributed protocol. In fact, distributed aspects of the system are ignored at this model level and the abstract model considers a centralised configuration. The following group of refinement steps ($m_{0..n}$) introduce more details about the model by primarily modelling communication aspects. For protocol modelling we propose to use backward unfolding style where the next refinement step introduces the preceding protocol step. The communication modelling patterns are primarily made of patterns for defining communicating actors of a system, introducing new messages into a model through context models (and machine variables) and machine event patterns for message exchanges. The previously developed communication modelling patterns and a generic model refinement plan are described in more detail in [SIA+19].

A key aspect of our methodology is the scenario validation and analysis. Particularly, in early protocol development stages, it might be too onerous to verify a model only to discover design mistakes. To facilitate design exploration we apply animation and model-checking enabled by ProB. Nonetheless, the final (concrete) model should be proved by adding invariants to the model and proving generated proof obligations with available automated theorem provers.

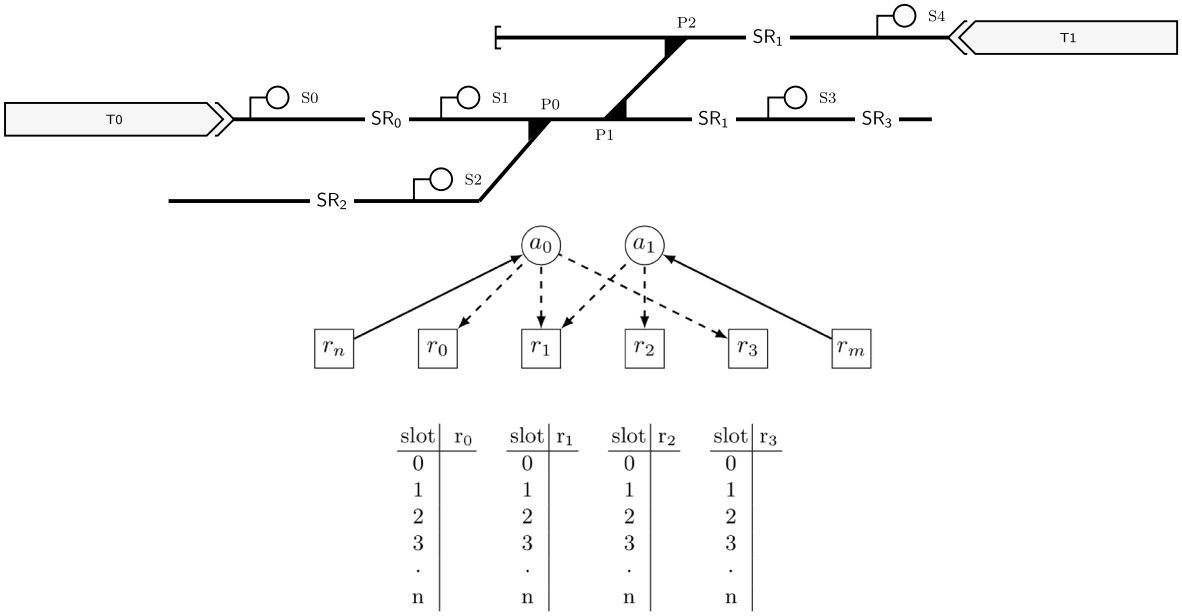

Fig. 5. An example of a distributed railway system and its abstract model: sub-route mapping to resources $\mathsf{SR}_0 \rightarrow \mathsf{r}_0, \cdots, \mathsf{SR}_3 \rightarrow \mathsf{r}_3$ and trains to agents mapping $\mathsf{T}_0 \rightarrow \mathsf{a}_0, \mathsf{T}_1 \rightarrow \mathsf{a}_1$. The bold arrows illustrate resources locked by agents while dashed arrows show resources which will need to be locked by an agent. For example, if train T_1 wants to travel to sub-route SR_2 it must simultaneously lock sub-routes SR_1 and SR_2.

Table 2. High-level distributed signalling system specifications

SYS_1	The distributed railway system is made of agents and resources (respectively represent trains and railway subsections).
SYS_2	Agents are only allowed to communicate with resources and not other agents.
SYS_3	Agents have an objective, which is a set of resources agents have to reserve before proceeding with the next objective.
SYS_4	Agents have a memory in which sent messages are stored.
SYS_5	Resources have a memory where agents allocation order can be stored.
SYS_6	Resources have a promised pointer (ppt) of the memory, which indicates a currently available resource allocation order.
SYS_7	Resources have a read pointer (rpt) of the memory, which specifies a memory slot (with an associated agent) that currently uses a resource.
ENV_1	Message delays are permitted in the distributed signalling system model.

3. High-level distributed signalling system model

We abstract the railway model and instead of trains, routes, and switches our system model consists of agents and resources (resources controllers). The system model permits message exchanges only between agents and resources, and messages can be delayed. Each resource controller has an associated queue-like memory, where the order of agent allocation can be stored. A resource also has a promise (ppt) and read pointers (rpt), which respectively indicate the currently available slot in the queue and the reserved slot (with an associated agent) that currently uses the resource (abstract railway model visualised in Fig. 5). An agent has an objective, which is a collection of resources an agent will attempt to reserve (all at the same time) before using and eventually releasing them. In Fig. 6 we visualise a simple resource locking protocol and illustrate how ppt and rpt variables are updated. In Table 2 high-level distributed system specifications are elicited in the format, which is suggested by the proposed methodology.

The main objective of the protocol is to enable safe and deadlock-free distributed atomic reservation of collection of resources. Where by a safe resource reservation we mean that no two different agents have reserved the same resource at the same time. The protocol must also guarantee that each agent eventually gets all requested resources-partial request satisfaction is not permitted. These are fundamental requirements of railway signalling systems which ensure a safe train separation by utilising a so-called fixed-block signalling [FH18].

Table 3. High-level system safety and liveness requirements

SAF_1	A resource will not be allocated to different agents at the same time.
SAF_2	An agent will not use a resource until all requested resources are allocated.
LIV_1	An agent must be eventually allocated requested set of resources.
LIV_2	Resource allocation must be guaranteed in the presence of message delays.

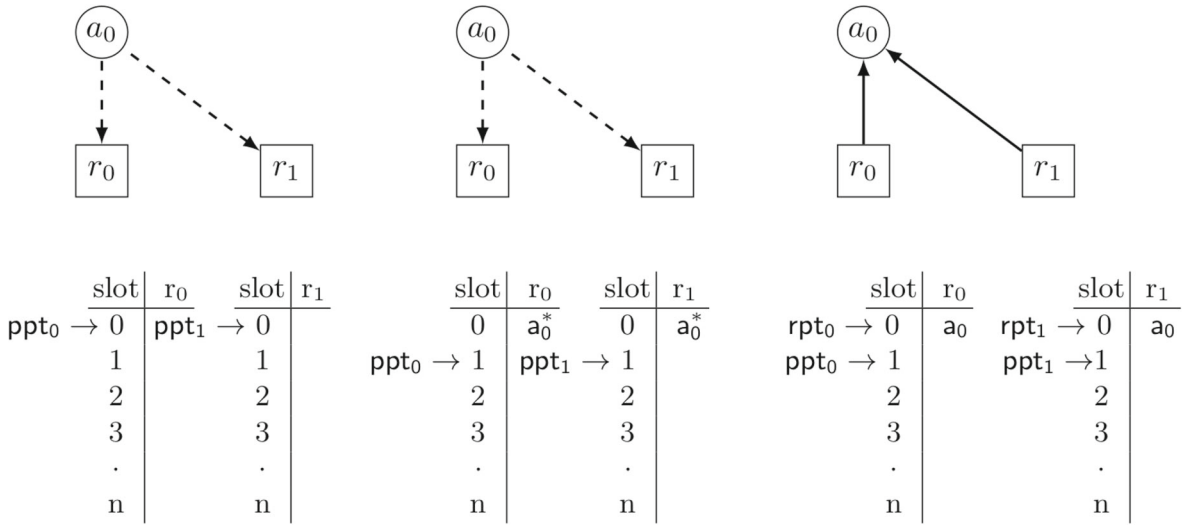

Fig. 6. A scenario (left to right) depicting a simple resource locking protocol with promised (ppt) and read (rpt) variable value changes.

The main high-level safety and liveness requirements of the distributed system are expressed in Table 3. The following section attempts to justify the need for an adequate distribute protocol by discussing problematic distributed resource allocation scenarios.

3.1. Problematic distributed resource allocation scenarios

Let us consider Scenarios 1-2 (visualised in Fig. 7) to see how requirement LIV_1 could not be guaranteed (while ensuring SAF_2) without an adequate distributed resource allocation protocol.

Scenario 1 In this scenario, agents a_0 and a_1 are attempting to reserve the same set of resources $\{\mathsf{r}_0, \mathsf{r}_1\}$. Agents start by firstly sending request messages to both resources. Once a resource receives a request message, it replies with the current value of the promised pointer $\mathsf{ppt}(\mathsf{r}_k)$ and then increments the $\mathsf{ppt}(\mathsf{r}_k)$. For instance, in this scenario, resource r_0 firstly received a request message from agent a_0 and thus replied with the value $\mathsf{ppt}(\mathsf{r}_0) = 0$, which was then followed by a message to a_1 with an incremented $\mathsf{ppt}(\mathsf{r}_0)$ value of 1. In Fig. 7, we denote a_n^* as the $\mathsf{ppt}(\mathsf{r}_k)$ value sent to a_n. Request messages at resource r_1 have been received and replied in the opposite order.

In this preliminary protocol, once an agent receives promised pointer values from all requested resources, it sends messages to requested resources to lock them at the promised queue-slot. In this scenario, agent a_0 was promised queue-slots $\{(\mathsf{r}_0, 0), (\mathsf{r}_1, 1)\}$ while a_1 queue-slots $\{(\mathsf{r}_0, 1), (\mathsf{r}_1, 0)\}$. If agents would lock these exact queue-slots, resource r_0 would allow a_0 to *use* it first, while r_1 would concurrently allow a_1. The distributed system would deadlock and fail to satisfy LIV_2 requirement as both agents would wait for the second *use* message to ensure SAF_2.

In order to prevent the cross-blocking type of deadlocks, an agent should repeatedly re-request the same set of resources (and not lock them) until all received promised queue slot values are the same. We define a process of an agent attempting to receive the same promised queue slots as an agent forming a distributed lane (dl).

Definition 1 A distributed lane dl of an agent a is a set of pairs $\mathsf{dl}(\mathsf{a}) = \{(\mathsf{r}_k, \mathsf{s}), \cdots, (\mathsf{r}_{k+m}, \mathsf{s})\}$ where $\{\mathsf{r}_k, \cdots, \mathsf{r}_{k+m}\}$ are all resources requested by an agent a and s is a natural number representing a slot value of the queue promised by all requested resources.

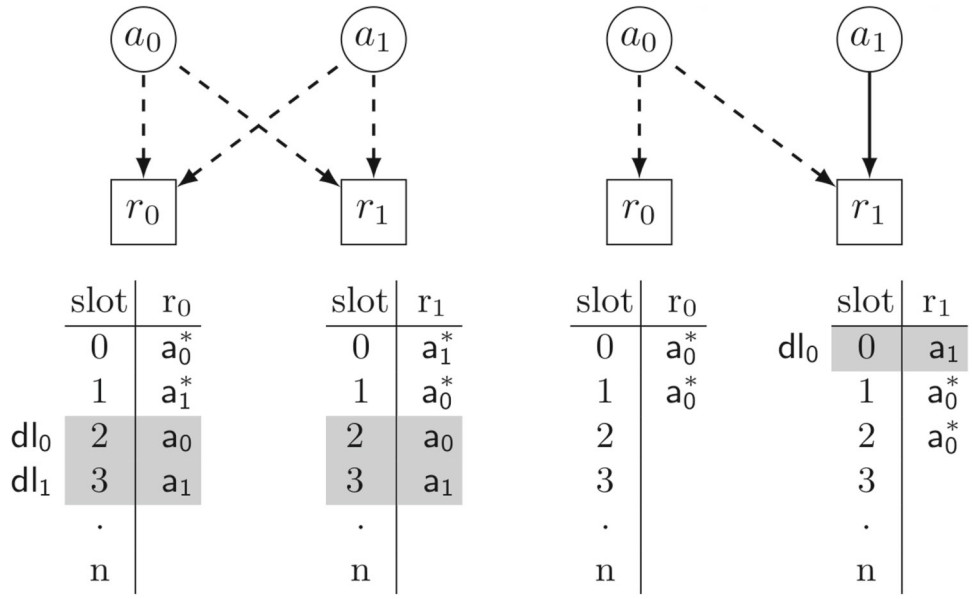

Fig. 7. Problematic scenarios: Scenario 1 (left) and Scenario 2 (right)

Important to note, that this solution relies on the assumption, that there is a non-zero probability of distinct messages arriving at the same destination in different orders, even if they are simultaneously sent by different sources.

The modified situation is depicted in Scenario 1, where, after agents $\{a_0, a_1\}$ initially receiving $\{(r_0, 0), (r_1, 1)\}$ and $\{(r_0, 1), (r_1, 0)\}$ slots, mutually re-request resources again. This time they receive $\{(r_0, 2), (r_1, 2)\}$ and $\{(r_0, 3), (r_1, 3)\}$ slots, and are able to form distributed lanes $dl_0(a_0)$ and $dl_1(a_1)$.

Scenario 2 However, simply re-requesting the same resources might result in a different problem. In Scenario 2, agent a_1 has requested and has been allocated a single resource r_1 which in turn modified $ppt(r_1)$ to 1 while $ppt(r_0)$ remained 0. If another agent a_0 attempts to reserve resources $\{r_0, r_1\}$, it will never receive the same promised pointer value from both resources per (re)-requesting round, and hence, will not be able to lock them.

3.2. Semi-formal protocol description

In order to address the issues described in the previous section, we developed a two-stage protocol, where the $stage_1$ of the distributed protocol specifies how an agent forms a distributed lane and $stage_2$ of the protocol addresses deadlock scenarios, which can occur after agents form distributed lanes. In the following paragraphs we semi-formally describe both stages of the protocol, which then will be formally modelled with the Event-B specification language (Algorithm 1—agent's algorithm for $stage_1$, Algorithm 2—resource's algorithm for both stages and Algorithm 3—agent's algorithm for $stage_2$).

$Stage_1$ An agent, which intends to reserve a set of resources starts by sending request messages to resources. The messages are sent to those resources which are part of the agent's current objective. In the provided pseudocode excerpt, we first denote relations sent_requests and objective where they are mappings from agents to resource collections (ln. 1-3 Algorithm 1). The messages request are sent by an agent a_n to a resource r_k ($r_k \in objective[a_n]$) until $sent\_requests[a_n] = objective[a_n]$ (images are equal) (ln. 4-7 Algorithm 1). When a resource r_k receives a request message from an agent a_n it responds with a reply message which contains the current promised pointer value of resource $ppt(r_k)$ to that agent and increments the promised pointer (ln. 6-8 Algorithm 2). After sending all request messages an agent waits until reply messages are received from requested resources (ln. 8 Algorithm 1) and then makes a decision.

If all received promised pointer values are the same (a distributed lane can be formed) an agent will complete $stage_1$ by sending write messages which contain the negotiated index to all requested resources (ln. 20-24 Algorithm 1). But if one of the received promised pointer values is different an agent will start a renegotiation cycle (ln. 9-19 Algorithm 1). An agent will now send a srequest messages which contain a desired slot index to resources.

Algorithm 1 Agent $stage_1$ communication algorithm

1: **variables** sent_requests, received_replies, sent_srequests, sent_write **typeof** AGT $\leftrightarrow$ RES **init** $\emptyset$
2: **variables** objective **typeof** AGT $\leftrightarrow$ RES **init** objective $:\in$ AGT $\nleftrightarrow$ RES
3: **variables** replies **typeof** AGT $\leftrightarrow$ $\mathbb{N}$ **init** $\emptyset$
4: **while** sent_requests[a_n] $\neq$ objective[a_n] **do** ▷ requesting resources which belong to the objective
5: request(a_n) $\rightarrow$ r_k
6: sent_requests := sent_requests $\cup$ {(a_n, r_k)}
7: **end**
8: **wait until** received_replies[a_n] = objective[a_n]
9: **while** | replies[a_n] |$\neq$ 1 **do** ▷ enter while loop if all received indices are not the same
10: sent_srequests[a_n] := $\emptyset$
11: received_replies[a_n] := $\emptyset$
12: replies[a_n] := $\emptyset$
13: **while** sent_srequests[a_n] $\neq$ objective[a_n] **do**
14: m := **max**(replies[a_n]) + 1
15: srequest(a_n, m) $\rightarrow$ r_k ▷ send a special request message with a desired slot index m
16: sent_srequests := sent_srequests $\cup$ {(a_n, r_k)}
17: **end**
18: **wait until** received_replies[a_n] = objective[a_n]
19: **end**
20: **while** sent_write[a_n] $\neq$ objective[a_n] **do**
21: m := **max**(replies[a_n])
22: write(a_n, m) $\rightarrow$ r_k
23: sent_write := sent_write $\cup$ {(a_n, r_k)}
24: **end**

A desired index is computed by taking the maximum of all received promised pointer values and adding a constant (one is sufficient) - ln. 14 Algorithm 1. A resource will reply to srequest message with the higher value of the current ppt(r_k) or received srequest message value and will update the promised pointer (ln. 9-11 Algorithm 2). After sending all srequest messages, an agent will wait for reply messages (ln. 18 Algorithm 1) and will restart the loop if received slot indices are not the same (ln. 9 Algorithm 1).

It is important to note that the stage 1 protocol solution to the described deadlock scenarios has a stochastic nature and one needs to guarantee that a desirable state is probabilistically reachable. In Table 4 we elicit additional safety and liveness requirements for the $stage_1$ of the protocol, which will need to be proved in the formal model. $Stage_2$ Once an agent completes sending all write messages it will wait for all pready messages from resources (ln. 4 Algorithm 3). A pready message is sent by a resource firstly if it has received a write message and no other agent is consuming that resource at that moment-resource is not locked (ln. 12–15 Algorithm 2). Secondly, a pready message will only be sent to an agent if it is distributed lane is the new minimum. In our protocol resources read pointer always take a new minimum value in the queue, once an agent sends a release message and allocation is removed from the queue.

Table 4. Low-level protocol stage_1 safety and liveness requirements

SAF_3	An agent will not send write (form a distributed lane) messages until all receive promised pointer values are identical.
SAF_4	Agents with overlapping resource objectives will negotiate distributed lanes with different index.
LIV_3	An agent will eventually negotiate a distributed lane.

Algorithm 2 Resource communication algorithm

```
1:  variables ppt typeof AGT → ℕ init RES → {0}
2:  variables rpt typeof AGT ↛ ℕ init ∅
3:  variables rlock typeof RES ↛ AGT init ∅
4:  variables mem typeof RES → (ℕ ↛ AGT) init RES ↔ {∅}
5:  switch received_message do
6:      case request(a_n)                          ▷ a resource replies to a request message with a slot index ppt(r_k)
7:          reply(ppt(r_k), r_k) → a_n
8:          ppt(r_k) := ppt(r_k) + 1
9:      case srequest(a_n, n)                      ▷ a resource replies to a special request message with a slot index ppt(r_k) or n
10:         reply(max(ppt(r_k), n), r_k) → a_n
11:         ppt(r_k) := max(ppt(r_k), n) + 1
12:     case write(a_n, m)                         ▷ a resource replies to a write message with a pready message if
13:         if rlock(r_k) = ∅ ∧ m = rpt(r_k)       ▷ a resource is not locked and read pointer is at slot m
14:             mem(r_k, m) := a_n
15:             pready(r_k) → a_n
16:     case lock(a_n)
17:         if rlock(r_k) = ∅                      ▷ a resource replies with a ready message if a resource is not locked
18:             rlock(r_k) := a_n                  ▷ a resource is locked and ready message is sent
19:             response(r_k, READY) → a_n
20:         if rlock(r_k) ≠ ∅
21:             response(r_k, DENY) → a_n          ▷ resource replies with a deny message if a resource is locked
22:     case release(a_n, m)                       ▷ a resource will unlock itself and remove an agent from memory slot m
23:         mem(r_k, m) := ∅
24:         rlock(r_k) := ∅
25:         if mem(r_k) ≠ ∅                        ▷ if memory is not empty a resource will
26:             rpt(r_k) := min(dom(mem(r_k)))     ▷ update a read pointer to the next reserved slot
27:             a_n := mem(r_k)(rpt(r_k))          ▷ and send pready message to that agent
28:             pready(r_k) → a_n
29:
```

When an agent receives all pready messages it will send lock messages to requested resources (ln. 5–8 Algorithm 3). If a resource is unlocked upon receiving lock message it will reply with ready message and lock itself, meaning, that it will stop sending pready messages (even to agent with smaller distributed lanes) until a release message is received from that agent (ln. 17–19 Algorithm 2). However, if, a resource is locked upon receiving lock message, it replies with deny message (ln. 20–21 Algorithm 2). If for every lock message an agent received a ready message, it can proceed to use resources and eventually release them. If, at least, one message is a deny message an agent will send release messages to resource (ln. 11–14 Algorithm 3) which sent ready messages to unlock them and will wait for pready messages again to repeat the process.

Algorithm 3 Agent $stage_2$ communication algorithm

```
1: variables received_response, received_pready, sent_lock, sent_release, consumed, released typeof AGT ↔ RES init ∅
2: variables response typeof RES ↔ (AGT ↔ {DENY, READY}) init ∅
3: while received_response[a_n] ≠ objective[a_n] do
4:    wait until received_pready[a_n] = objective[a_n]          ▷ wait until all pre-ready messages are received
5:    while sent_lock[a_n] ≠ objective[a_n] do                  ▷ attempt to lock all requested resources
6:       lock(a_n) → r_k
7:       sent_lock := sent_lock ∪ {(a_n, r_k)}
8:    end
9:    wait until received_response[a_n] = objective[a_n]        ▷ wait until all ready messages are received
10:   if DENY ∈ ran(response)[a_n] do
11:      while sent_release[a_n] ≠ response^-1[(a_n, DENY)] do  ▷ resources which sent DENY message
12:         release(a_n) → r_k
13:         sent_release := sent_release ∪ {(a_n, r_k)}
14:      end
15:      received_response[a_n] := ∅
16:      received_pready[a_n] := ∅
17:      sent_lock[a_n] := ∅
18:      sent_release[a_n] := ∅
19:   end
20: end
21: while consumed[a_n] ≠ objective[a_n] do                     ▷ all locked resources (stage_2 completed) are consumed
22:    consumed := consumed ∪ {(a_n, r_k)}
23: end
24: while released[a_n] ≠ objective[a_n] do                     ▷ all consumed resources are eventually released
25:    release(a_n) → r_k
26:    released := released ∪ {r_k, a_n}
27: end
```

4. Formal protocol development with Event-B

We apply the Event-B formalism to develop a high-fidelity functional model and prove the protocol functional correctness requirements. We follow the modelling process presented in Section 2. Important to note that the protocol model was redeveloped multiple times as various deadlock scenarios were found with ProB animator and model-checker. In following sections we present the final (verified) model.

4.1. A refinement strategy of the distributed protocol

The model development approach we utilise is model refinement, which starts with the abstract model that formally specifies the objective of the protocol. In fact distributed aspects of the system are ignored at this model level and the abstract model considers a centralised configuration. The abstract resource allocation protocol model was captured by two machines (m_0 and m_1). The former model essentially summarises the high-level objective of the protocol which is agents safely capturing and releasing collection of resources (objectives). This abstract model contains individual events for capturing and releasing objectives. The next refinement step introduces resources into the model and decomposes two previously introduced events according to the loop modelling pattern defined in [SIA+19].

The following group of refinement steps introduces more details about the model by primarily modelling aspects of communication. For protocol modelling, we propose to use the backward unfolding style where the next refinement step introduces the preceding protocol step. The abstract models were firstly refined with stage_2 segment of the protocol. In the refinement, m_2, we introduced lock, response and release messages and associated events into the model. In this step we also demonstrated that the protocol stage_2 ensures safe distributed resource reservation by proving an invariant. The invariant states that no two agents will be both at resource consuming stage if both requested intersecting collections of resources. The following refinement, m_3, is the bridge between protocol stages stage_1 and stage_2 and introduces two new messages write and pready into the model. In the final refinement step (m_4) we model stage_1 of the distributed protocol which is responsible for creating distributed lanes. The remaining messages request, reply, srequest and associated events are introduced together with the distributed lane data structure. In this refinement, we prove that distributed lanes are correctly formed.

Abstract Model Context The abstract model context introduces agents, resources and objectives into the model as finite carrier sets. The context also contains the enumerated set for the agent's status.

Message Context All messages in the protocol were defined in individual context models according to the communication pattern presented in [SIA+19].

Abstract Model The initial machine (abstract model) summarises the purpose of the protocol. The high-level objective of the protocol is to facilitate the reservation of collection of resources (objectives) by agents. The abstract model captures an agent reserving and eventually releasing an objective.

Refinement 1 (Abstract ext.) In this refinement we introduce resources into the model and now an agent tries to fulfil the objective by locking individual resources (and releasing). To model the iterative process of locking/releasing resources we use loop modelling pattern.

Refinement 2 Because of the backward unfolding style the model is then refined with stage_2 part of the protocol. In the refinement lock, response and release messages are introduced. With this refinement step we also demonstrate that the protocol's stage_2 ensures safe distributed resource reservation by proving an invariant. The invariant states that no two agents will be both at resource consuming stage if both requested intersecting collections of resources.

Refinement 3 This refinement can be considered as a bridge between the protocol's stages stage_1 and stage_2. Here, two new messages—write and pready—are introduced into the model.

Refinement 4 The final refinement step captures the stage_1 of the distributed protocol which is responsible for creating distributed lanes. The remaining messages request, reply, srequest and associated events are introduced together with the distributed lane data structure. In this refinement, we prove that distributed lanes are correctly formed (req. SAF_{3-4}).

CONTEXT context_abstract
SETS
AGT, RES, OBJ ▷ agents, resources and objectives
CONSTANTS
objr ▷ a constant function which maps objectives to a set of resources (axm_7)
AXIOMS
$axm_{1..3}$ finite(AGT, RES, OBJ)
$axm_{4..6}$ $\emptyset \subset$ RES, AGT, OBJ
axm_7 objr $\in$ OBJ $\rightarrow \mathbb{P}1$(RES)
axm_8 $\exists o \cdot o \in dom(objr) \Rightarrow card(objr(o)) \geq 1$

Listing 1. A context of the abstract resource allocation model

CONTEXT context_agent_state
SETS
AST ▷ agents, resources and objectives
CONSTANTS
REQUEST, WRITE, RENEGOTIATE, CONFIRMR, CONSUME, RELEASE, CONFIRMW,
LOCK, CONFIRMC, UNLOCK, CONFIRMP
AXIOMS
axm_1 partition(AST, {REQUEST}, {CONFIRMW}, {WRITE}, {RENEGOTIATE},
{CONFIRMR}, {CONFIRMP}, {LOCK}, {UNLOCK}, {CONFIRMC}, {CONSUME}, {RELEASE})

Listing 2. A context model defining program counter values of an agent

4.2. Protocol Event-B model: abstract context

The formal protocol modelling was started by defining static model information such as carrier sets, constants and axioms. First of all we create a context for the abstract model which contains three finite size carrier sets representing agents (AGT), resources (RES) and objectives (OBJ) as shown in Listing 1. The latter carrier set is used as an extractor operator for groups of resources in a constant function (objr).

We also introduce an enumerated set (AST) to denote agent status or in other words agent's program counter values in a separate context model context_agent_state (Listing 2). For the abstract model only (CONSUME) and (RELEASE) elements are needed whereas remaining elements will be introduced in the following subsections.

4.3. Protocol Event-B model: machine m_0

In modelling the distributed resource allocation protocol we follow a standard Event-B modelling approach where the abstract model summarises the protocol with a centralised view of the system. As previously discussed, the objective of the distributed protocol is to enable safe resource locking. This can be abstracted as agents consuming and releasing objectives (visualised in Fig. 8). In the abstract model we want to prevent agents consuming identical objectives.

To begin with, we introduce two variables for storing consumed objectives (cons) and agents status (pct0) as shown in Listing 3. The agent consume event updates the variable cons with a new pair (act_1 in Listing 4) if an objective has not been consumed (guard grd_1) and an agent is not consuming any other objectives (guard grd_2). In addition the event updates agent's program counter-variable which helps to track the steps of the agent and discharge proof obligations (guard act_2).

The second event (Listing 5) models the release of an objective by removing a pair which belonged to the cons variable and updating the program counter. The correctness of the abstract model can be verified by proving invariant (inv_{saf} in Listing 3) which asks to prove that an objective is only consumed by a single agent. The Rodin platform automatically generates proof obligations, like one below, which requires to prove that an invariant inv_{saf} is preserved by the agent_consume event.

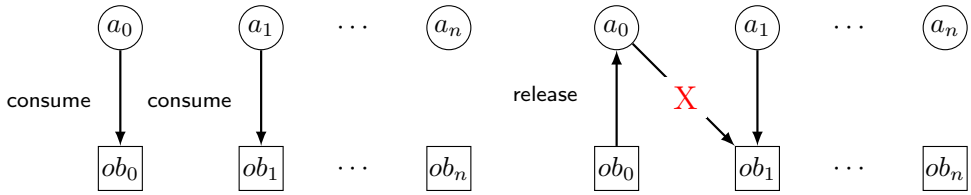

Fig. 8. A visualisation of the abstract resource allocation model (m_0): model consists of agents which consume free objectives and then release them

Guards (and other invariants, axioms) of the agent_consume event become hypothesis of the proof obligation and if they are *strong* enough make it is possible for available automatic theorem provers to prove them automatically. A similar proof obligation was generated and automatically proved for the agent_release event.

$$
\begin{array}{ll}
cons^{-1} \in OBJ \nrightarrow AGT & (\mathsf{inv_{saf}}) \\
obj \in OBJ \setminus ran(cons) & (\mathsf{event : agent\_consume : grd_1}) \\
ag \in AGT \setminus dom(cons) & (\mathsf{event : agent\_consume : grd_2}) \\
pct0(ag) = CONSUME & (\mathsf{event : agent\_consume : grd_3}) \\
\vdash & \\
(cons \lhd\!\!\leftarrow \{ag \mapsto ob\})^{-1} \in OBJ \nrightarrow AGT &
\end{array}
$$

4.4. Protocol Event-B model: machine m_1

The refinement step m_1 expands on the previous refinement by introducing resources into the model. Instead of simply consuming an objective, an agent captures resources until an objective is fulfilled (visualised in Fig. 9). Captured resources are stored in newly created variable capt whereas objective an agent will try to complete in the function objt (shown in Listing 6).

In contrary to capturing a single objective, an agent might need to consume multiple resources in order to fulfill its objective. For the iterative process, we previously introduced a two event pattern which we instantiate in this refinement step for capturing and releasing events (see Section IV.D in [SIA+19]). The loop body event agent_consume_b (Listing 7) takes any agent with previously initialised objective and assign a new resource rs to the variable capt (action act_1 in Listing 7). The agent must be at CONSUME state (guard grd_5), the resources must be within agent's objective (guard grd_2) and not yet be captured by any agent (guard grd_3).

MACHINE m_0
SEES context_abstract
VARIABLES
cons, pct0
INVARIANTS
inv_1 $\mathsf{cons} \in \mathsf{AGT} \nrightarrow \mathsf{OBJ}$ ▷ variable which maps agents to consumed objectives
inv_2 $\mathsf{pct0} \in \mathsf{AGT} \rightarrow \mathsf{AST}$ ▷ variable which maps agents to its current program counter status
$\mathsf{inv_{saf}}$ $\mathsf{cons}^{-1} \in \mathsf{OBJ} \nrightarrow \mathsf{AGT}$ ▷ safety invariant which requires an objective to be consumed by only a single agent

Listing 3. The machine m_0 of the distributed resource allocation model

EVENT agent_consume $\widehat{=}$
ANY
ag, ob ▷ agent (ag) and objective (ob)
WHERE
grd_1 ob ∈ OBJ \ ran(cons) ▷ take an objective that has not been consumed
grd_2 ag ∈ AGT \ dom(cons) ▷ take an agent that has not consumed an objective
grd_3 pct0(ag) = CONSUME ▷ program counter of the agent (ag) must be CONSUME
THEN
act_1 cons(ag) := ob ▷ assign (consume) an objective (ob) to an agent (ag)
act_2 pct0(ag) := RELEASE ▷ update program counter
END

Listing 4. Events modelling an agent consuming an objective

EVENT agent_release $\widehat{=}$
ANY
ag, ob
WHERE
grd_1 ag ∈ dom(cons) ▷ take an agent that has consumed an objective
grd_2 ob ∈ cons(ag) ▷ objective consumed by agent ag
grd_3 pct0(ag) = RELEASE
THEN
act_1 cons := cons \ {ag ↦ ob} ▷ remove agent and its consumed objective
act_2 pct0(ag) := CONSUME ▷ update program counter
END

Listing 5. Events modelling an agent releasing resources an objective

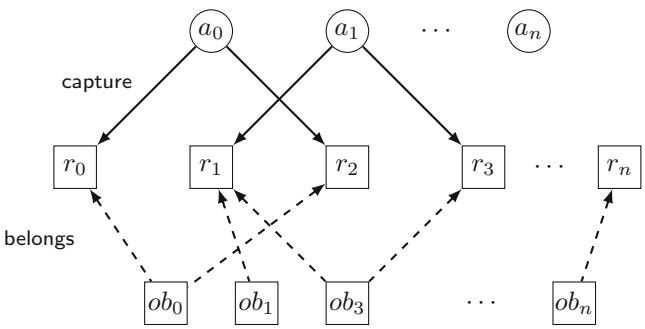

Fig. 9. A visualisation of the (m_1) refinement model: agents are capturing free resources to fulfil an objective (a resource can belong to different objectives)

MACHINE m_1
REFINES m_0
VARIABLES
capt, objt, pct1
INVARIANTS
inv_1 $capt \in AGT \rightarrow \mathbb{P}(RES)$ ▷ a variable mapping agents to captured resources
inv_2 $objt \in AGT \rightarrow OBJ$ ▷ variable mapping an agent to an objective it tries to fulfil
⋮
inv_{saf} $\forall a_1, a_2 \cdot a_1 \in dom(capt) \land a_2 \in dom(capt) \land a1 \neq a2 \Rightarrow capt(a1) \cap capt(a2) = \emptyset$

Listing 6. Variables and invariants of the refinement step m_1

EVENT agent_consume_b $\widehat{=}$
ANY
ag, rs ▷ agent (ag) and resource (rs)
WHERE
grd_1 $ag \in dom(capt)$ ▷ take some agent (ag) from domain of variable captured
grd_2 $rs \in objr(objt(ag))$ ▷ take a resource (rs) which is part of ag objective (objt(ag))
grd_3 $rs \notin union(ran(capt))$ ▷ resource must not be captured by other agent
grd_4 $union(ran(\{ag\} \lhd capt)) \cap objr(objt(ag)) = \emptyset$ ▷ no resources from the objective can be captured
grd_5 $pct1(ag) = CONSUME$ ▷ program counter of agent must be at CONSUME
THEN
act_1 $capt(ag) := capt(ag) \cup \{rs\}$ ▷ add resource (rs) to agent's captured set
END

Listing 7. The event modelling an agent consuming a free resource: loop body event

The agent ag will also capture a resource rs only if none of the resources from its objective have been captured by other agents (guard grd_4). The guard grd_4 prevents a deadlock scenario in which agents with overlapping objectives are unable to complete their objective as remaining resources are captured by the other overlapping agent.

The loop completion event agent_consume_c (Listing 8) would be triggered as soon as the objective has been fulfilled (guard grd_1 in Listing 8) and program counter would be updated to new state—RELEASE (action act_1 in Listing 8). Similarly, in this refinement we transform agent_release event according based on the event pattern presented. To show correctness of the extended model, we prove an invariant (inv_{saf} in Listing 6) which states that no two agents can have the same resource captured.

EVENT agent_consume_c $\widehat{=}$
ANY
ag
WHERE
grd_1 $capt(ag) = objr(objt(ag))$ ▷ guard which checks wheter resource obective has been fulfilled
grd_2 $pct1(ag) = CONSUME$
THEN
act_2 $pct0(ab) := RELEASE$ ▷ update agent's program counter to RELEASE
END

Listing 8. The event modelling an agent consuming a free resource: loop completion event

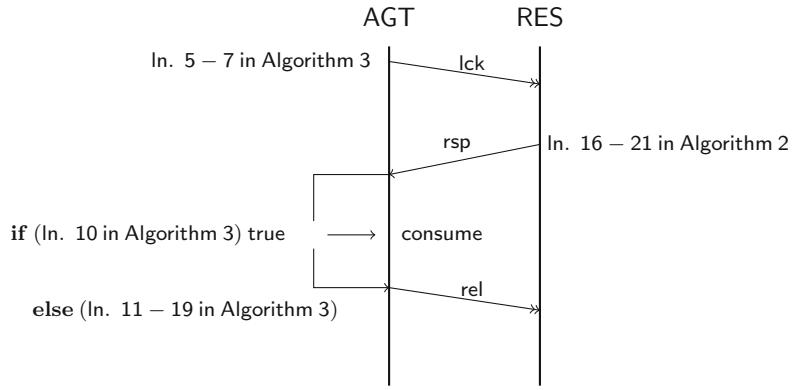

Fig. 10. Sequence diagram of stage_2 of the protocol at refinement level m_2

4.5. Protocol Event-B model: machine m_2

In this refinement step, we start to consider communication as we introduce protocol (described in Section 3) into the model. Because of backward unfolding style we introduce stage_2 part of the protocol first (sequence diagram shown in Fig. 10). At this stage of the protocol three types of messages are sent: lock and release by an agent and response message by a resource. In this refinement step, we apply communication modelling patterns described in Section IV.C of the paper [SIA+19].

MACHINE m_2
REFINES m_1
VARIABLES
rdpt, lck, lcke, rsp, rel, pct2
INVARIANTS
inv_1 $\mathsf{rdpt} \in \mathsf{RES} \nrightarrow \mathsf{AGT}$ ▷ variable representing locks of resources (resource locked by an agent)
inv_2 $\mathsf{lck} \subseteq \mathsf{LCK}$ ▷ lock message channel variable
inv_3 $\mathsf{lcke} \in \mathsf{AGT} \rightarrow \mathbb{P}(\mathsf{RES})$
inv_4 $\mathsf{rsp} \subseteq \mathsf{RSP}$ ▷ response message channel variable
inv_5 $\mathsf{rel} \subseteq \mathsf{REL}$ ▷ release message channel variable
inv_6 $\mathsf{pct2} \in \mathsf{AGT} \rightarrow \mathsf{AST}$

Listing 9: Variables and invariants of the refinement step m_2

First of all, machine m_2 is extended with three context models-separately defining each message type according to the generic message context pattern (response message context model shown in Listing 12). Then, by following machine communication modelling pattern we create variables lck, rsp and rel which represent communication channels of each message. In addition an agent's variable lcke is created to store already sent lck messages (shown in Listing 9).

In the stage_2 an agent tries to lock resources associated with negotiated distributed lane by sending lock messages. To model that, we apply loop modelling patterns and create two new events: agent_lock_b and agent_lock_c (Listings 10 and 11). Since, preceding protocol messages are modelled in the next refinements, we model lock message as a initiating message at this stage but later convert it to a reply event type. The first event is the body of a message sending loop which sends a new lock message (action act_1 in Listing 10) if a message lc has not been sent before (guard grd_1) and destination (resource) is within agent's objective (guard grd_3). The program counter of the agent must be in LOCK protocol phase (guard grd_4). An agent also saves a local copy of the sent message with the event action act_2.

The second event in the loop pattern is a loop completion event agent_lock_c which detects the end of the loop and updates the program counter (Listings 11). For the lock message sending event—an agent must detect when all messages have been sent or in other words the objective has been fulfilled (guard grd_2).

EVENT agent_lock_b $\widehat{=}$
ANY
lc ▷ lc (lock) message which will be sent
WHERE
grd_1 lc ∈ LCK \ lck ▷ only a new lock message will be sent (lc message not already in the lck channel)
grd_2 lckd(lc) ∉ lcke(lcks(lc))
grd_3 lckd(lc) ∈ objr(objt(lcks(lc))) ▷ a lock message is sent to a resource which belongs to agent's objective
grd_4 pct2(lcks(lc)) = LOCK ▷ lock message can only be sent by an agent with LOCK program status
THEN
act_1 lck := lck ∪ {lc} ▷ a new lock message is added to lock message channel
act_2 lcke(lcks(lc)) := lcke(lcks(lc)) ∪ lckd(lc) ▷ message sending receipt is saved locally by an agent
END

Listing 10. The event modelling an agent locking resources: loop body event

EVENT agent_lock_c $\widehat{=}$
ANY
ag
WHERE
grd_1 ag ∈ dom(lcke)
grd_2 lcke(ag) = objr(objt(ag)) ▷ lock messages are sent to all resources which belong to agent's objective
grd_3 pct2(ag) = LOCK
THEN
act_1 pct2(ab) := CONFIRMC ▷ once all lock messages are sent agent's program counter is updated
END

Listing 11. The event modelling an agent locking resources: loop completion event

This event simply updates the program counter to CONFIRMC state with the action (action act_1). According to the protocol, after sending all lock messages an agent will wait for replies before it proceeds to consuming resources.

In order to model a resource's response to the lock message first we created a read pointer variable rdpt to work as a resource lock for an agent. The resource lock is released once a resource receives a release message from an agent who locked the resource (see Listing 9). A resource sending a response message is a reply type message therefore we used a reply event modelling pattern (Listings 13 and 14). A resource sends response message when lock message has been received-guard grd_1 in both events. The following two guards (grd_4 and grd_5) define the source and destination of the new message which are respectively destination and source of received lock message.

CONTEXT
resource_response_message
SETS
RSP, CNF
CONSTANTS
rsps, rspd, rspv, READY, DENY
AXIOMS
axm_1 rsps ∈ RSP ⇸ RES ▷ source constant function of the response message
axm_2 rspd ∈ RSP ⇸ AGT ▷ destination constant function of the response message
axm_3 rspv ∈ RSP ⇸ CNF ▷ value constant function of the response message
axm_4 partition(CNF, {READY}, {DENY}) ▷ response message value can be READY or DENY
axm_5 finite(RSP)
axm_6 ∀s, d, v · s ∈ RES ∧ d ∈ AGT ∧ v ∈ CNF ⇒ ∃m · rsps(m) = s ∧ rspd(m) = d ∧ rspv(m) = v

Listing 12: The context model of the resource response type message

Lastly, the response message also carries a value and guards grd_6 define the value in both events. The message carries READY value if the resource is not locked by other agent (grd_3 in Listing 13) otherwise a DENY message is sent (grd_3 in Listing 14). In addition to sending a message, the resource_response_ready event also removes answered message and if READY message was sent resource locks itself for that agent with action act_3.

EVENT resource_response_ready $\widehat{=}$
ANY
  lc, rs ▷ lock (lc) and response (rs messages)
WHERE
  grd_1 lc ∈ lck ▷ a response READY message will be only sent if a lock message was was sent
  grd_2 rs ∈ RSP \ rsp ▷ only a previously not sent response READY message will be sent
  grd_3 rsps(rs) ∉ dom(rdpt) ▷ a response READY message is sent by a resource only if it is not locked
  grd_4 rsps(rs) = lckd(lc) ▷ source of the response message is destination of received lock message
  grd_5 rspd(rs) = lcks(lc) ▷ destination of the response message is source of received lock message
  grd_6 rspn(rs) = READY ▷ response message contains READY value
THEN
  act_1 rps := rsp ∪ {rs} ▷ response message is added (sent) to response channel
  act_2 lck := lck \ {lc} ▷ received and replied lock message is removed from the channel
  act_3 rdpt := rdpt ⩤ {rsps(rs) ↦ rspd(rs)} ▷ resource which sent READY response message is locked for the agent
END

Listing 13: The event modelling a resource sending a response message: READY type

EVENT resource_response_deny $\widehat{=}$
ANY
  lc, rs
WHERE
  grd_1 lc ∈ lck ▷ a response DENY message will be only sent if a lock message was was sent
  grd_2 rs ∈ RSP \ rsp
  grd_3 rsps(rs) ∈ dom(rdpt) ▷ a response DENY message is sent by a resource if it is locked
  grd_4 rsps(rs) = lckd(lc)
  grd_5 rspd(rs) = lcks(lc)
  grd_6 rspn(rs) = DENY ▷ response message contains DENY value
THEN
  act_1 rps := rsp ∪ {rs} rps := rsp ∪ {rs} ▷ response message is added (sent) to response channel
  act_2 lck := lck \ {lc} lck := lck \ {lc} ▷ received and replied lock message is removed from the channel
END

Listing 14: The event modelling a resource sending a response message: DENY type

Once an agent sends all lock messages it must receive all response messages before it can progress. In the model, we created a new event agent_decide (Listing 15) which checks the values of received response messages and selects the new program counter value. In principle this event is a loop completion event but in the model we decided to model *decision* events separately to reduce events complexity. The main difference from the basic loop completion event is that depending on conditions different program counter values can be selected. In agent_decide event if all messages contained READY value (guard grd_2) then the event will update program counter to CONSUME. But if a DENY message (guard grd_3) was received program counter is changed to UNLOCK state. In the latter scenario an agent must then send release messages to resources, which sent READY messages and locked their resources, so protocol can progress.

For selecting correct message from the channel we use messages constant functions and a channel variable—$[rsp \cap rspd^{-1}[\{ag\}]$ selects all rsp messages which were sent to agent ag. Inserting the result to the rsps constant function we get sources (resources) of these messages. This message extraction guard pattern is used widely when relevant messages need to be selected. Events of the unlocking phase are modelled using reply communication pattern and hence not covered in this subsection. Once relevant resources are unlocked agent tries to lock them again in this refinement step.

EVENT agent_decide $\widehat{=}$
ANY
ag, pc ▷ agent (ag) and program counter (pc)
WHERE
grd_1 rsps[rsp ∩ rspd^{-1}[{ag}]] = lcke(ag) ▷ all response messages have to be received
grd_2 rspn[rsp ∩ rspd^{-1}[{ag}]] = {READY} ⇒ pc = CONSUME ▷ if all response messages are READY
grd_3 DENY ∈ rspn[rsp ∩ rspd^{-1}[{ag}]] ⇒ pc = UNLOCK ▷ if at least one response message is DENY
grd_4 pct2(ag) = CONFIRMC
THEN
act_1 pct2(ag) := pc ▷ update agent's program counter based on guards 2 − 3
END

Listing 15. The event modelling agents decision between restarting $stage_2$ or consuming resources

INVARIANTS
SAF_1 ∀ a1, a2 · pct2(a1) = CONSUME ∧ pct2(a2) = CONSUME ∧ a1 ≠ a2
⇒ objr(objt(a1)) ∩ objr(objt(a2)) = ∅

Listing 16. The safety invariant for prohibiting mutual resource locking by different agents

In this refinement step, which specifies $stage_2$ of the distributed protocol, we were required to prove invariant expressed in Listing 16. The safety invariant relates to the mutual exclusion safety protocol property SAF_1 specified in Fig. 3. The property requires proving that if any two agents are in the CONSUME state their objectives (set of resources) must not be overlapping. The Rodin platform has generated five proof obligations based from the SAF_1 invariant. Four out of five proof obligations were proved automatically, however, the safety invariant relating to SAF_1 property and agent_decide event had to be proved interactively. The interactively discharged proof obligation is shown below, where the guards and invariant form the hypothesis of the proof obligation while the invariant modified by the agent_decide event action is goal of the sequent.

$rsps[rsp \cap rspd^{-1}[\{ag\}]] = lcke(ag)$ (grd_1)
$rspn[rsp \cap rspd^{-1}[\{ag\}]] = \{READY\} \Rightarrow pc = CONSUME$ (grd_2)
$\forall a_1, a_2 \cdot (pct2)(a_1) = CONSUME \wedge pct2(a_2) = CONSUME \wedge a_1 \neq a_2 \Rightarrow$ (inv_{saf})
$objr(objt(a_1)) \cap objr(objt(a_2)) = \emptyset$
$\vdash$
$\forall a_1, a_2 \cdot (pct2 \leftarrow \{ag \mapsto pc\})(a_1) = CONSUME \wedge (pct2 \leftarrow \{ag \mapsto pc\})(a_2) = CONSUME \wedge a_1 \neq a_2 \Rightarrow$
$objr(objt(a_1)) \cap objr(objt(a_2)) = \emptyset$

4.6. Protocol Event-B model: machine m_3

In this refinement step we continue to unfold protocol backwards and introduce two new messages write and pready. In fact this refinement step can be thought as an intermediate step gluing protocol stages $stage_1$ and $stage_2$ (sequence diagram shown in Fig. 11). Events for capturing write message sending were developed using the initiating message pattern. Since they were structurally identical to the previous refinement events agent_lock_b and agent_lock_c events we do not discuss them here. The more interesting challenge of this refinement step was correctly capturing the pready message. A variable of type mem ∈ RES → ℙ(AGT) was created to abstract queue like resource memories (see Listing 17) which are introduced in the next refinement step.

A resource sends pready message to inform an agent of its availability for consumption and that it can be locked now. There are two different cases when this message should be sent and instead of constructing a single event to cover all scenarios it was decided to create more trivial events for each of the cases.

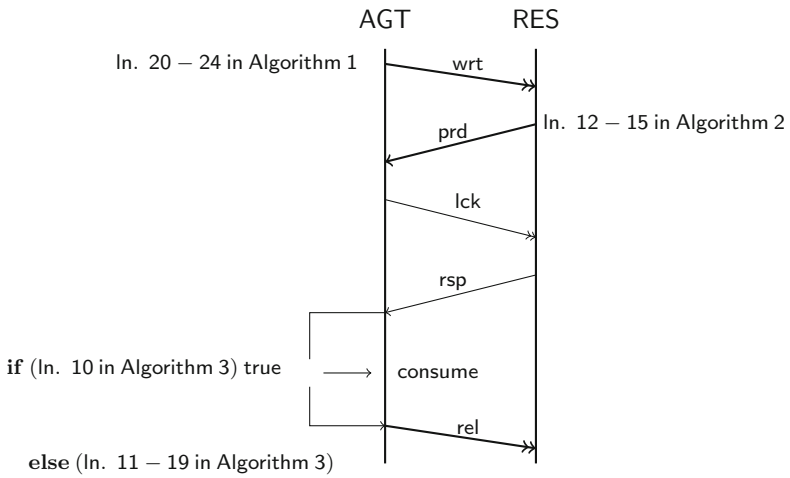

Fig. 11. Sequence diagram of stage_1 of the protocol at refinement level m_3

MACHINE m_3
REFINES m_2
VARIABLES
wrt, prd, mem, wrte, pct3
INVARIANTS
inv_1 $\mathsf{wrt} \subseteq \mathsf{WRT}$ ▷ write message channel variable
inv_2 $\mathsf{prd} \subseteq \mathsf{PRD}$ ▷ pready message channel variable
inv_3 $\mathsf{mem} \in \mathsf{RES} \rightarrow \mathbb{P}(\mathsf{AGT})$ ▷ resource memory variable storring agents which have sent write message

Listing 17. Variables and invariants of the machine model m_3

In the model we created a reply type resource_write_preready event which creates a new message and removes the answered message from the wrt channel if a resource is not locked. A pready message will be sent if a write message has been received and resource is not locked by an agent.

The second event resource_release_pready was necessary to cover a scenario where pready message was sent to an agent in response to its write message but in the end the agent was not able to lock all resources. This would mean that write messages have been removed and resource_write_preready guards would never be satisfied for that agent. Yet an agent would be still waiting to receive pready messages eventually. Therefore we introduced initiating message resource_release_pready event which sends a new pready message to any agent which is interested in that resource if a resource is not locked.

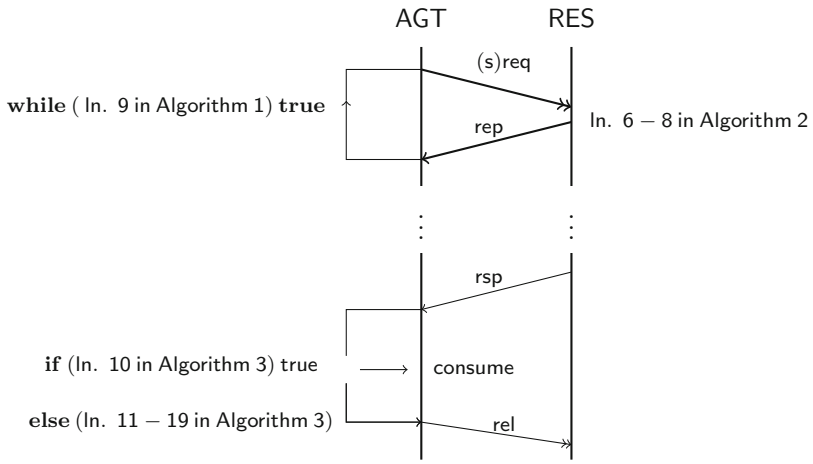

Fig. 12. Sequence diagram of stage_1 of the protocol at refinement level m_4

4.7. Protocol Event-B model: machine m_4

The final refinement step m_4 introduces the remaining part of the protocol—stage_1. This stage starts with an agent requesting a set of resources and finishes with agent forming a distributed lane by sending a write messages to the same resources. In between, an agent might need to send a special request message srequest if a distributed lane was not negotiated in the first attempt. To allow distributed lane forming—a resource memory introduced in the previous refinement now must be refined to queue-like data structure with corresponding variables (e.g. read pointer, promised pointer).

MACHINE m_4
REFINES m_3
VARIABLES
req, rqst, rqs, rqts, rep...
INVARIANTS
inv_1 $\mathsf{req} \subseteq \mathsf{REQ}$ ▷ request message channel
inv_2 $\mathsf{rqst} \in \mathsf{AGT} \rightarrow \mathbb{P}(\mathsf{RES})$
inv_3 $\mathsf{rqs} \subseteq \mathsf{REQ}$ ▷ special request message channel
inv_4 $\mathsf{rqts} \in \mathsf{AGT} \rightarrow \mathbb{P}(\mathsf{RES})$
inv_5 $\mathsf{rep} \subseteq \mathsf{REP}$ ▷ reply message channel
inv_6 $\mathsf{ppt} \in \mathsf{RES} \rightarrow \mathbb{N}$ ▷ promised pointer variable
inv_7 $\mathsf{rpt} \in \mathsf{RES} \nrightarrow \mathbb{N}$ ▷ read pointer variable
inv_8 $\mathsf{lan} \in \mathsf{RES} \rightarrow (\mathbb{N} \nrightarrow \mathsf{AGT})$ ▷ variable where distributed lanes are stored
inv_9 $\mathsf{pct4} \in \mathsf{AGT} \rightarrow \mathsf{AST}$

Listing 18: Variables and invariants of the machine model m_4

To begin with we start with introducing communication channel variables for request, reply and srequest messages. In addition two local variables rqst and rqts were created to store messages on the agent side (shown in Listing 18). Following the loop and initiating message modelling patterns we introduced two events agent_request_b and agent_request_c for requesting resources (Listings 19 and 20).

```
EVENT agent_request_b ≙
ANY
    rq
WHERE
    grd1  rq ∈ REQ \ req    ▷ send a new request message
    grd2  reqd(rq) ∉ rqst(reqs(rq))
    grd3  reqd(rq) ∈ objr(objt(reqs(rq)))    ▷ request message destination (resource) must be part of the objective
    grd4  pct4(reqs(rq)) = REQUEST
THEN
    act1  req := req ∪ {rq}    ▷ send a request message
    act2  rqts(reqs(rq)) := rqts(reqs(rq)) ∪ {reqd(rq)}
END
```

Listing 19: The event modelling an agent requesting a resource: loop body event

After all request messages have been sent event agent_request_c changes agents program counter to CONFIRMW state (act_1 in Listing 20) which informs an agent to wait until all reply messages have been received.

```
EVENT agent_request_c ≙
ANY
    ag
WHERE
    grd1  ag ∈ dom(rqst)
    grd2  rqst(ag) = objr(objt(ag))    ▷ check if all request messages have been sent
    grd3  pct4(ag) = REQUEST
THEN
    act1  pct4(ag) := CONFIRMW    ▷ update program counter value
END
```

Listing 20: The event modelling an agent requesting a resource: loop completion event

On the resource side, a reply type event resource_reply_general (Listing 21) was created which simply sends a new reply message (action act_1) containing the current promised pointer value (guard grd_5), removes request message (act_2) and increments the promised pointer (act_3). Once an agent has sent all request and respectively received all reply messages a decision must be made whether to form a distributed lane or renegotiate new indexes.

```
resource_reply_general ≙
ANY
    rp, rq    ▷reply message (rp) and request message (rq)
WHERE
    grd1  rq ∈ req
    grd2  rp ∈ REP \ rep
    grd3  repd(rp) = reqs(rq)
    grd4  reps(rp) = reqd(rq)
    grd5  repn(rp) = ppt(reqs(rq))
THEN
    act1  rep := rep ∪ {rp}    ▷ send a reply message
    act2  req := req \ {rq}    ▷ remove request message which wasreplied
    act3  ppt(reps(rp)) := ppt(reps(rp)) + 1    ▷ update (increment) resource's promised pointer value
END
```

Listing 21: The event modelling a resource replying to a EVENTrequest message

The distributed lane will be formed if all reply messages contained the same promised pointer value otherwise an agent must renegotiate a distributed lane. In the model, we created a new *decision* agent_confirm_write_renegotiate event which changes the program counter according to the named conditions (Listing 22). To check whether all reply messages contained the same index it is sufficient to check the set size (cardinality) of the

repn[rep ∩ repd^{-1}[ag]] values set. The guard grd$_1$ must have both local variables because this event can be enabled after general request or srequest message sending.

EVENT agent_confirm_write_renegotiate ≙
ANY
ag, pc
WHERE
grd$_1$ ag ∈ repd[rep]
grd$_2$ card(rqst(ag) ∪ rqts(ag)) = card(rep ∩ repd^{-1}[{ag}]) ▷ all sent request messages have been replied
grd$_3$ reps[rep ∩ repd^{-1}[{ag}]] = objr(objt(ag))
grd$_4$ pct4(ag) = CONFIRMW
grd$_5$ card(repn[rep ∩ repd^{-1}[{ag}]]) > 1 ⇒ pc = RENEGOTIATE ▷ renegotiate resources
grd$_6$ card(repn[rep ∩ repd^{-1}[ag]]) = 1 ⇒ pc = WRITE ▷ form a distributed lane if all reply values identical
THEN
act$_1$ pct4(ag) := pc
END

Listing 22: Agent decision that decides whether to restart stage$_1$ or start stage$_2$

If an agent was not able to negotiate a distributed lane it must send a new, special message—srequest. This time instead of requesting an arbitrary index a new srequest message contains a desired index which is computed by adding a non-zero constant to the highest reply index. To send srequest messages we created two reply type events: agent_renegotiate_b and agent_renegotiate_c.

As a standard reply type event agent_renegotiate_b creates a new message and removes the answered reply message from the channel. Whereas agent_renegotiate_c event updates the program counter again to CONFIRMW state - this cycle repeats until a distributed lane is negotiated.

Lastly in this refinement we introduce distributed lane data structure and accordingly update relevant events. Two variables ppt and rpt are created to respectively represent promise pointer and read pointer of a resource. For each resource the promised pointer is initialised to zero and only two reply events resource_reply_general and resource_reply_special (Listing 23) modify a promised pointer variable. The first event simply increments the current ppt(r) value after reply message has been sent whereas the latter event updates the ppt(r) by computing action act$_3$, where maximum function parameters are the current ppt(r) value and received srequest message value.

EVENT resource_reply_special ≙
ANY
rp, rq ▷ reply message (rp) and special request message (rq)
WHERE
grd$_1$ rq ∈ rqs
grd$_2$ rp ∈ REP \ rep
grd$_3$ repd(rp) = rqss(rq)
grd$_4$ reps(rp) = rqsd(rq)
grd$_5$ repn(rp) = max({ppt(reps(rp)), rqsn(rq)}) ▷ reply message value is a larger value between resources ppt value and value from received special request message
THEN
act$_1$ rep := rep ∪ {rp}
act$_2$ rqs := rqs \ {rq}
act$_3$ ppt(reps(rp)) := max({ppt(reps(rp)), rqsn(rq)}) + 1 ▷ update resource's promised pointer
END

Listing 23: The event modelling a resource replying to a srequest message

Read pointers are updated by two events which send pready messages. In contrary to the previous variable the read pointer rpt(r) is always set to the minimum value of the request pool. This is necessary as some agent might negotiate a distributed lane with lower index than others but its write messages are delayed (or even lost) so the protocol would halt. Allowing agents with higher distributed lane indexes but sooner write message arriving to consume resources introduces fault-tolerance into the protocol. Guards of these two events were also strengthened to only send pready messages to newly incoming agents if their index is the lowest in the request pool.

INVARIANTS

$$\mathsf{SAF}_3 \quad \forall ag \cdot pct4(ag) = \mathsf{WRITE} \wedge reps[rep \cap repd^{-1}[\{ag\}]] = objr(objt(ag)) \Rightarrow card(repn[rep \cap repd^{-1}[ag]]) = 1$$

Listing 24. Event-B safety invariant ensuring that SAF_3 requirement is preserved in the model

MACHINE m_4
VARIABLES

$req, \cdots, his_{ppt}, his_{wr}$

INVARIANTS

$$\vdots$$
$$his_{ppt} \in RES \rightarrow (\mathbb{N} \nrightarrow \mathbb{N})$$
$$his_{wr} \in RES \rightarrow \mathbb{N}$$
$$inv_{saf4} \quad \forall r, n_1, n_2 \cdot r \in RES \wedge n_1, n_2 \in dom(his_{ppt}(r)) \wedge n_1 < n_2 \Rightarrow his_{ppt}(r)(n1) < his_{ppt}(r)(n2)$$
$$inv\_his_{ppt} \quad \forall res \cdot (his_{wr}(res) = 0 \wedge his_{ppt}(res) = \emptyset) \vee (dom(his_{ppt}(res)) = 0 \,..\, his_{wr}(res) - 1 \wedge his_{ppt}(res)(his_{wr}(res) - 1) = ppt(res) - 1)$$

Listing 25. Additional variables and invariants added for provind safety property SAF_4

Proving SAF_{3-4} Safety Properties. As shown in Section 3.2 (Scenarios 1 - 2) high-level system requirements can only be met if an agent invariably and correctly forms a distributed lane. The probabilistic lane forming eventuality (LIV_3) is discussed in [SIK+20] while in the following paragraphs we describe proving requirements SAF_{3-4} in the Event-B model.

First of all, requirement SAF_3 (see Table 4) relates to SAF_2 (see Table 3) and is needed to ensure that that an agent is not allocated only a part of resources it has requested. In the model we ensure that SAF_3 requirement is satisfied by strengthening event guards which restrict enabling states of the event that generates an outgoing write message. To cross-check this implementation we add an invariant that directly shows that SAF_3 is maintained in the model (Listing 24).

In the following paragraphs we provide details on proving a slightly more interesting case of SAF_4 which assumes that SAF_3 property is proven. Requirement SAF_4 addresses potential cross-blocking deadlock scenarios described in Section 3.1 or a resource double locking due to distributed lane overriding. The strategy to proving the SAF_4 requirement is to formally prove that agents with intersecting objectives (at least one common resource) always form distributed lanes with differing indices. Based on a distributed lane definition (Definition 1 in Section 3.1) we could formally express the safety invariant as expression (1) which states if two different agents (a_1, a_2) have formed a distributed lane (dl) over an intersecting set of resources ($\mathbf{dom}(dl(a_m)) = \{r_0, r_1, ..r_n\}$) then slot values s of their distributed lanes must be different.

$$\forall a_1, a_2 \cdot a_1, a_2 \in AGENTS \wedge a_1 \neq a_2 \wedge \mathbf{dom}(dl(a_1)) \cap \mathbf{dom}(dl(a_2)) \neq \emptyset \Rightarrow \mathbf{ran}(dl(a_1)) \cap \mathbf{ran}(dl(a_2)) \neq \emptyset \tag{1}$$

Instead of directly including predicate (1) into the Event-B model and completing a complex interactive SAF_4 property proof, it was realised that property (1) can be expressed (and proved) as a simpler verification condition. Let's first assume that agents only form distributed lanes if all received indices are the same (proved as SAF_3). If the same resource is requested by different agents and they are replied with different promised pointer values, these promised pointer values (slots) will be distributed lane *deciders* as all other slots from remaining resources must be the same in order to form a distributed lane. Therefore, to prove that model preserves SAF_4 requirement it is enough to show that a resource always replies to a request or special request message with a unique promised pointer value. We can prove that a resource always replies with a unique promised pointer value by introducing a history variable into our model.

To prove this property we first extend our model with a variable his_{ppt} and an entry variable his_{wr} or *time-stamp* as shown in Listing 25. The former variable stores each resources chronological promised pointer updates and latter works as a write pointer for history variable.

EVENT resource_reply_general $\widehat{=}$
ANY
rq, rp, res
WHERE
grd_1 $rq \in req$
⋮
grd_6 $res = reps(rp)$
THEN
act_1 $rep := rep \cup \{rp\}$ ▷ send a reply message
⋮
act_4 $his_{ppt}(res) := his_{ppt}(res) \leftarrowtail \{his_{wr}(res) \mapsto ppt(res)\}$ ▷ update history ppt variable
act_5 $his_{wr}(res) := his_{wr}(res) + 1$ ▷ update history write variable
END

Listing 26. The event modelling a resource replying to a request message with history variable extension

Table 5. Event-B protocol model proof statistics

Model	No. of POs	Aut. Discharged	Int. Discharged
context c_0	0	0	0
context mes.	9	9	0
machine m_0	12	12	0
machine m_1	23	21	2
machine m_2	59	43	16
machine m_3	43	32	11
machine m_4	103	57	46
Total	249	174	75

After introducing history variables, we updated two events resource_reply_{general, special} which update promised pointer variables (after sending reply messages) by adding two new actions act_4 and act_5 as shown in Listing 26 (identical extension for resource_reply_special event).

Action act_4 updates a history variable for a resource res with the current write stamp and promised pointer (ppt(res)) value sent. The next action act_5 simply updates the resource's write stamp. We can then add the main invariant to prove (inv_{saf4} in Listing 25) which states that if we take any two entries n1, n2 of the history variable for the same resource where one is larger, then that larger entry should have larger promised pointer value.

To prove that resource_reply_{general, special} preserve inv_{saf4}, the following properties play the key role: (1) the domain of his_{ppt} (i.e., 'indices' of his_{ppt}) is $\{0, \ldots, his_{wr} - 1\}$, (2) $his_{ppt}(his_{wr} - 1) < his_{ppt}(his_{wr})$. Property (2) holds because $his_{ppt}(his_{wr})$ is the maximum of promised pointer (ppt) and special request slot number and promised pointer is incremented as resource_reply_{general, special} occurs. We also specified these properties as an invariant (inv_his_{ppt}) and proved they are preserved by the events which helped to prove inv_{saf_4}.

Proof statistics. In Table 5 we provide an overall proof statistics of the Event-B protocol model which may be used as a metric for models complexity. The majority of the generated proof obligations were automatically discharged with available solvers and even a large fraction of interactive proofs required minimum number of steps. We believe that a high proof automation was due to modelling patterns [SIA+19] use and the SMT-based verification support [ISAYR16, DFGV14].

5. Conclusions and future work

In this paper, we demonstrated how a previously proposed formal development methodology of distributed systems can be used to developing a distributed signalling system protocol. Starting only with high-level system requirements, we developed an early formal protocol prototype model, which with the help of ProB model checker was refined as subtle deadlock scenarios were discovered. Because of the stepwise modelling principle and our refinement strategy, re-modelling efforts were significantly reduced as only a part of the complete functional Event-B model needed to be reworked. For example, issues discovered in the stage_1 of the distributed protocol only affected final refinement step m_4 and so machine models $m_{0..3}$ and associated (completed) proofs were preserved. The stepwise distributed protocol development, as also shown before [CM06, HKBA09, ILTR11], together with adequate tools [ISAYR16, DFGV14] helped to achieve fairly high verification automation. The final protocol model was formally proved to guarantee a safe and deadlock free distributed signalling operation. We demonstrated this by expressing and deductively proving invariants which stated that a distributed lane would be correctly and eventually formed by an agent. An alternative deadlock-freedom proving strategy would have been proving a deadlock freedom proof obligation (page 5 DLK [Hoa14]) which was considered in our work. The resulting proof obligation results in a complex proof obligation to be discharged in the Rodin platform.

In this work, we have also demonstrated that the previously developed communication modelling patterns [SIA+19] can model a variety of message sending/receiving scenarios, and therefore, provide a more systematic communication modelling method. On the other hand, the proposed formal development methodology does not provide a systematic process to refine the generic communication-based signalling model. Formalising refinement steps would fully utilise the benefits of the Event-B stepwise modelling and further reduce modelling and verification effort.

We believe that the future work of defining refinement patterns can be built around some initial refinement ideas in our paper [SIA+19]. Furthermore, the modelling effort with our methodology could be significantly reduced by providing an automatic refinement of communication models from a high-level protocol specification, for example, sequence diagrams, which are a widely used method to specify protocols in the railway industry. Another important direction we would like to explore in the future work is protocol's viability to be deployed in practice. At the moment the distributed resource allocation protocol solves railway resource allocation problem at a high abstraction level and in the future work the protocol model should be refined to consider other railway signalling elements such as switches, signals and routes as in [HP00]. The refined version of the protocol would also need to solve another type of deadlock scenario which would occur once railway routes are introduced into the model. Specifically, the head-to-head deadlock scenario, which can occur when two trains cannot progress as each train is preventing the other one from locking the next set of resources (example discussed in Section 7.1.4 of [GH21]). In the future work we could address this issue by extending the protocol to specify how objectives are formed and selected. Finally, in the future work, we would like to prototype a physical distributed signalling system with the extended distributed resource allocation protocol could be implemented by using, for example the CLEARSY Safety Platform [LDFO20] as suggested in paper [Pro16].

Acknowledgements

This work is supported by an iCASE studentship and funded by EPSRC/UK and Siemens Rail Automation (grant number EP/P510580/1). This work was partially supported by the EPSRC STRATA platform grant (EP/N023641/1).

References

[Abr96] Abrial J R (1996) The B-book: assigning programs to meanings. Cambridge University Press, New York

[Abr13] Abrial JR (2013) Modeling in Event-B: system and software engineering. Cambridge University Press, New York

[Bac90] Back RJR (1990) Refinement calculus, part II: parallel and reactive programs. In: de Bakker JW, de Roever WP, Rozenberg G (eds) Stepwise refinement of distributed systems models, formalisms, correctness. Springer, Berlin, pp 67–93

[BBFM99] Behm P, Benoit P, Faivre A, Meynadier JM (1999) Météor: a successful application of B in a large project. In Wing JM, Woodcock K, Davies J (eds) FM'99—formal methods. Springer, Berlin, pp 369–387

[BSR80] Bernstein PA, Shipman DW, Rothnie JB Jr (1980) Concurrency control in a system for distributed databases (SDD-1). ACM Trans Database Syst 5(1):18–51

[CM06] Cansell D, Méry D (2006) Formal and incremental construction of distributed algorithms: on the distributed reference counting algorithm. Theor Comput Sci 364(3):318–337

[DFGV14] Déharbe D, Fontaine P, Guyot Y, Voisin L (2014) Integrating SMT Solvers in Rodin. Sci Comput Program 94(P2):130–143

[ED06] Essamé D, Dollé D (2006) B in large-scale projects: the Canarsie line CBTC experience. In: Julliand J, Kouchnarenko O (eds) B 2007: formal specification and development in B. Springer, Berlin, pp 252–254

[EGLT76] Eswaran Kapali P, Gray Jim, Lorie Raymond A, Traiger Irving L (1976) The Notions of Consistency and Predicate Locks in a Database System. Commun. ACM, 19(11):624–633

[FH18] Fantechi A, Haxthausen AE (2018) Safety interlocking as a distributed mutual exclusion problem. In Howar F, Barnat J (eds) Formal methods for industrial critical systems. Springer, Cham, pp 52–66

[FHN17] Fantechi A, Haxthausen AE, Nielsen MBR (2017) Model checking geographically distributed interlocking systems using UMC. In: 25th Euromicro international conference on parallel, distributed and network-based processing (PDP), pp 278–286

[GH21] Geisler S, Haxthausen AE (2021) Stepwise development and model checking of a distributed interlocking system using RAISE. Formal Aspects Comput 33:87–125

[GR92] Gray J, Reuter A (1992) Transaction processing: concepts and techniques, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco

[HHK+17] Hawblitzel C, Howell J, Kapritsos M, Lorch JR, Parno B, Roberts ML, Setty S, Zill B (2017) IronFleet: proving safety and liveness of practical distributed systems. Commun ACM 60(7):83–92

[HKBA09] Hoang TS, Kuruma H, Basin D, Abrial JR (2009) Developing topology discovery in Event-B. Sci Comput Program 74(11):879–899

[HKNP06] Hinton A, Kwiatkowska M, Norman G, Parker D (2006) PRISM: a tool for automatic verification of probabilistic systems. In: Hermanns H, Palsberg J (eds) Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 441–444

[Hoa14] Hoang TS (2014) Reasoning about almost-certain convergence properties using Event-B. Sci Comput Program 81:108–121

[HP00] Haxthausen AE, Peleska J (2000) Formal development and verification of a distributed railway control system. IEEE Trans Softw Eng 26(8):687–701

[ILTR11] Iliasov A, Laibinis L, Troubitsyna E, Romanovsky A (2011) Formal derivation of a distributed program in Event B. In: Qin S, Qiu Z (eds) Formal methods and software engineering. Springer, Berlin, pp 420–436

[ISAYR16] Iliasov A, Stankaitis P, Adjepon-Yamoah D, Romanovsky A (2016) Rodin platform Why3 plug-in. In: Proceedings of the 5th international conference on abstract state machines, alloy, B, TLA, VDM, and Z, ABZ 2016. Springer, Berlin, pp 275–281

[Pro16] INTO-CPS Project (2016) Deliverable D1.2-Case studies 2. Available at https://into-cps.org/publications/

[LB03] Leuschel M, Butler M (2003) ProB: a model checker for B. In: Araki K, Gnesi S, Mandrioli D (eds) FME 2003: formal methods. Springer, Berlin, pp 855–874

[LDFO20] Lecomte T, Déharbe D, Fournier P, Oliveira M (2020) The CLEARSY safety platform: 5 years of research, development and deployment. Sci Comput Program 199:102524

[Mor96] Morley MJ (1996) Safety assurance in interlocking design. PhD thesis, University of Edinburgh. College of Science and Engineering. School of Informatics

[New14] Newcombe C (2014) Why Amazon Chose TLA+. In: Ait Ameur Y, Schewe KD (eds) Abstract state machines, alloy, B, TLA, VDM, and Z. Springer, Berlin, pp 25–39

[SDS+19] Stankaitis P, Dupont G, Singh NK, Ait-Ameur Y, Iliasov A, Romanovsky A (2019) Modelling hybrid train speed controller using proof and refinement. In: 2019 24th International conference on engineering of complex computer systems (ICECCS), pp 107–113

[SI17] Stankaitis P, Iliasov A (2017) Theories, techniques and tools for engineering heterogeneous railway networks. In: Fantechi A, Lecomte T, Romanovsky A (eds) Reliability, safety, and security of railway systems. Modelling, analysis, verification, and certification. Springer, Cham, pp 241–250

[SIA+19] Stankaitis P, Iliasov A, Ait-Ameur Y, Kobayashi T, Ishikawa F, Romanovsky A (2019) A refinement based method for developing distributed protocols. In: IEEE 19th international symposium on high assurance systems engineering (HASE), pp 90–97

[SIK+20] Stankaitis P, Iliasov A, Kobayashi T, Aït-Ameur Y, Ishikawa F, Romanovsky A (2020) Formal distributed protocol development for reservation of railway sections. In: Raschke A, Méry D, Houdek F (eds) Rigorous state-based methods. Springer, Cham, pp 203–219

[The06] The RODIN platform (2006). Available at https://sourceforge.net/projects/rodin-b-sharp/files/Core_Rodin_Platform/

[WK12] Whitwam F, Kanner A (2012) Control of automatic guided vehicles without wayside INterlocking. Patent US 20120323411 A1

Received 21 January 2021

Accepted in revised form 30 September 2021 by Alessandro Fantechi, Anne Haxthausen and Jim Woodcock