A Refinement Based Method for Developing Distributed Protocols

Paulius Stankaitis, Alexei Iliasov, Yamine Aït-Ameur, Tsutomou Kobayashi, Fuyuki Ishikawa, Alexander Romanowski

A refinement based method for developing distributed protocols

Paulius Stankaitis*, Alexei Iliasov*, Yamine Aït-Ameur†, Tsutomu Kobayashi‡,
Fuyuki Ishikawa‡ and Alexander Romanovsky*

*Newcastle University, Newcastle upon Tyne, United Kingdom
†INPT-ENSEEIHT, 2 Rue Charles Camichel, Toulouse, France
‡National Institute of Informatics, Tokyo, Japan
Corresponding author: p.stankaitis@newcastle.ac.uk

***Abstract*—This paper presents a methodology for modelling and verification of high-assurance distributed protocols. In the paper we describe two main technical contributions needed for the development method: communication modelling patterns and a refinement strategy. The applicability of the proposed method is demonstrated by developing a new distributed resource allocation protocol. We also discuss the necessity of integrating other tools such as stochastic model checkers for enabling verification of wider range of protocol properties.**

I. Introduction

Developing distributed systems is an intricate process, which requires rigorous methods due to concurrent nature of the distributed systems. Formal methods - mathematical model driven techniques - provide a systematic approach for developing complex systems. They offer an approach to specify systems precisely via mathematically defined syntax and semantics as well as formally validate them by using semi-automatic or automatic verification tools. In particular formal notations such as Event-B [1] are thought to be well suited for development and verification of various protocols. The stepwise and proof driven development provided by such methods is attractive to developers and can significantly reduce the modelling and verification effort.

The overall aim of our research is to reduce complexity of applying formal methods for developing high-assurance distributed protocols. The proposed development methodology is based on a stepwise refinement and mathematical proof - two techniques which we believe can reduce modelling effort and provide a higher system assurance degree. Nonetheless, the effectiveness of these techniques significantly correlates with developers experience and an adequate tool support. Thus, to overcome these issues our research focuses on developing a method for automatically constructing model refinement chains from high-level distributed protocol specifications and integrating a better tool support for animation and verification of the model. In this paper we discuss the current main elements developed to facilitate this approach: communication modelling patterns and a generic refinement strategy. The paper also presents how these patterns can be applied by developing a new distributed resource allocation protocol. We also discuss the endured verification challenges and results from using additional verification techniques.

Related work. There have been several studies which aimed to develop a general way of modelling and verification of distributed protocols. But, not many methods were based on the refinement, we discuss ones that utilized the step wise development. In Iliasov et al. [6] authors presented a modelling technique based to bridge the gap between Event-B formal model and the software protocol implementation. Their approach proposes to introduce an environment into the model and to further decompose communication events to separately model sending and receiving a message. The paper [8] proposed an integrated method based on Event-B and BIP [9] modelling languages for development of distributed systems. Their approach helps for a developer to interactively refine an abstract centralised system model with an assistance of domain languages and available plug-ins, and generate BIP model code. In the work by Hawblitzel et al. [10] authors developed a methodology to model and verify a non-trivial distributed systems (including implementations). Their method relies on proving refinement relation between different layers (e.g. design level and implementation) with a well known techniques such as TLA+ and state-of-the-art SMT solvers.

	r_0	r_1	r_2	r_3	
dl_0	0	0	0	0	
	1	1	1	1	
dl_1	2	2	2	2	dl_2
dl_3	3	3	3	3	
	4	4	4	4	dl_4
	5	5	5	5	

Fig. 1: An example virtual distributed lane data structure with 5 agents and 4 resources. Each dl_n only belongs to a single agent. Multiple distributed lanes can have the same index, but they cannot overlap (e.g. dl_1 and dl_2).

DOI 10.1109/HASE.2019.00023

In Section II we introduce the problem of a distributed resource allocation and propose a two stage distributed protocol for solving it. The following section describes our proposed development methodology in more detail including the background information on the Event-B specification language, ProB model checker [7] and PRISM model checker [5]. Section IV describes developed modelling patterns and refinement strategy to facilitate our development approach. In Section V we discuss how modelling patterns, refinement strategy and external tools were used to prove the distributed resource allocation protocol. The last section contains a conclusion and overviews the future work.

II. Overview of the Protocol

The objective of the protocol is to enable safe distributed atomic reservation of a collection of resources. For instance any two distinct agents A_0 and A_1 may require any resource collections r_0 and r_1, where $r_0, r_1 \subseteq R$. The protocol must guarantee that each agent gets all or nothing - partial request satisfaction is not permitted - and ensure that every agent request will be eventually satisfied as long as certain degenerated situations are avoided. A resource itself has an attributed memory, where requests can be stored together with a read pointer $\mathsf{rp}(r_k)$, and a promise pointer $\mathsf{pp}(r_k)$ - the largest promised index in the r_k request pool. In our system setting a resource is only allowed to exchange messages with agents and agents only with resources. Even though, we do not consider degenerate or malicious situations (messages cannot be altered or lost) requests can arrive at resources in any order. Permitting situations where requests can arrive in any order can cause situations where different requests are blocking each other and cause the system to deadlock.

In order to prevent a system deadlock and ensure progress we offer a two stage solution. The principal mechanism of solution we offer is distributed lane - a virtual data structure, which is only present at a conceptual level. To be more specific, a distributed lane is a uniform (single index) horizontal slice through request pools where a request pool is a resources request memory (vertical structure). A unique distributed lane can only belong to a single agent. An example of virtual distributed lane data structure is shown in Fig. 1. To lock (form a lane) resources, an agent has to go through a number of steps. The agents distributed protocol side can be split into two stages.

Stage 1 In this stage an agent attempts to negotiate a distributed lane by first sending request messages to resources of interest (objective). Once a resource receives a request message it replies with the current promised pointer value. An agent must wait to receive all reply messages before the decision is made. If all received promised pointer indexes are the same, an agent will create a distributed lane by sending write message. Otherwise, an agent will attempt to negotiate a distributed lane again by sending a special request message, which contains a desired value - lines 7–10 in Algorithm 1. The negotiation process continues until a lane is negotiated. Because of a probabilistic renegotiation nature we demonstrate in Section V that an agent will eventually negotiate a distributed lane.

Algorithm 1 Agent communication algorithm

```
1:  while sent_requests[A_n] ≠ objective[A_n] do
2:      request(A_n) → r_k    Sending message from A_n to r_k.
3:  end
4:  wait until received_replies[A_n] = objective[A_n]
5:  if |replies[A_n]| ≠ 1    All replied indexes are not the same.
6:    then while |replies[A_n]| ≠ 1 do
7:            m = max(replies(A_n)) + 1
8:            while sent_srequests[A_n] ≠ objective[A_n] do
9:                srequest(A_n, m) → r_k
10:           end
11: while sent_write[A_n] ≠ objective[A_n] do
12:     write(A_n, m) → r_k
13: end    The end of stage_1 of the protocol.
14: wait until received_pready[A_n] = objective[A_n]
15: while sent_lock[A_n] ≠ objective[A_n] do
16:                 lock(A_n) → r_k
17: end
18: wait until received_response[A_n] = objective[A_n]
19: if deny ∈ responses[A_n] then    Exists a deny message.
20:     forall r_k ∈ received_response(responses[A_n] ◁ ready)
21:                 release(A_n) → r_k
22:     end
23:     repeat from line 13
24: else consume resources
25: while sent_releases[A_n] ≠ objective[A_n] do
26:     release(A_n) → r_k
27: end
```

Stage 2 This stage begins when an agent negotiates its distributed lane and was mainly introduced when subtle unsafe scenarios were discovered by animating the initial model. After sending write messages, an agent must wait until it receives all pready messages. Once all messages have been received, an agent will try to lock resources by sending lock messages. If between sending pready and receiving lock messages a resource has not received other lock sooner, that resource will send a response(ready) message. Once an agent receives all response messages it will make another decision. If all an agent received all response(ready) messages, that agent can proceed an consume resources. If at least one of the messages was response(ready) that agent will send release messages to resources, which sent response(ready) and will repeat the process by again waiting for pready all messages.

The resource in this protocol only replies to the agents messages, therefore its communication can be described with switch-case pseudocode (see Algorithm 2). Updating resource read pointers is perhaps the most interesting element in this algorithm part. In contrary to the promised pointer, the read pointer $\mathsf{rpt}(r)$ is always set to the minimum value of the request pool. This is necessary as an 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. Important to note that a resource removes distributed lanes once a release message has been received so the read pointer value would change.

ALGORITHM 2 Resource communication algorithm

```
1:  switch received_message do
2:      case request(A_n)
3:          reply(ppt_k, r_k) → A_n
4:          ppt(r_k)' = ppt(r_k) + 1
5:      case srequest(A_n, n)
6:          reply(max(ppt_k, n) + 1, r_k) → A_n
7:          ppt(r_k)' = max(ppt_k, n) + 1
8:      case write(A_n, n)
9:          if free(r_k) ∧ n = min(req_pool(r_k)) then
10:             pready(r_k) → A_n
11:             rpt(r_k)' = n
12:     case lock(A_n)
13:         if free(r_k) then
14:             response(ready, r_k) → A_n
15:             lock(r_k, A_n)
16:         else response(deny, r_k) → A_n
17:     case release(A_n)
18:         req_pool(r_k) = req_pool(r_k) − req(A_n)
19:         if A_m · dist_lane(A_m) = min(req_pool(r_k)) then
20:             unlock(r_k)
21:             pready(r_k) → A_m
22:             rpt(r_k)' = min(req_pool(r_k))
```

III. INTEGRATED METHODOLOGY

This section overviews the proposed stepwise refinement and proof based development methodology of distributed protocols. We also discuss how this methodology was used to model and verify the distributed resource allocation protocol. Indeed, the stepwise refinement modelling approach would allow to manage a high distributed protocol modelling complexity by model abstraction and decomposition. The stepwise model development approach can also significantly reduce the mathematical proving effort which we regard as essential for ensuring safety of high-assurance systems. In our research we work towards an automatic refinement chain generation from high-level distributed protocol specification. In this paper we present key elements of the method: generic communication modelling patterns and the refinement strategy for modelling various distributed protocols. Our methodology also proposes to integrate other available tools for animating and model checking the distributed protocol model.

The cornerstone of the proposed methodology is a well known modelling framework - Event-B/Rodin. The Event-B method is a proof-based modelling language which facilitates a gradual system design through the stepwise refinement. The developer incrementally adds more detail in the proceeding refinement steps by including new information or proving properties. Properties are manually inserted as invariants and must be preserved by all state transitions (invariant preservation rule). The method we propose would automatically generate a distributed protocol model refinement chain from a high-level specification notation. For that we developed Event-B communication modelling patterns and a refinement strategy presented in the following section with an example. The generated Event-B model would then have multiple uses due to a number of useful plug-ins developed for the Rodin Platform including ProB animator and model checker.

In developing the distributed resource allocation protocol we extensively used the ProB model checker and its animator for an early development stage validation. The ProB model checker together with built-in constraint solver enables an iterative model exploration for possible correctness violations or deadlocks. These tools allowed to discover subtle deadlock scenarios and hence modify protocol specifications before mathematically proving the model. Once the model developer has sufficient confidence about the model correctness, it must be proved by defining safety invariants and proving them. The paper contains one of the correctness proofs completed for distributed resource allocation protocol. However, not all properties are easy to define, in particular, for a distributed protocol verification.

In our example the distributed resource allocation protocol had a stochastic nature. Probabilistic or liveness properties are hard to formalise and prove in the Event-B method. Therefore, it was decided to prove progress of the protocol by redeveloping part of the model in the PRISM model checker. The PRISM model checker is a well established symbolic model checking tool, which allows to model and analyse probabilistic systems. Several types of stochastic input models are supported but predominately discrete-time Markov chains (DTMC), continuous-time Markov chains (CTMC) and Markov decision processes (DTMC) are used. The tool has also functionality for result visualisation which was used to observe protocols performance as the number of agents and resources were increased. In general, we believe that our methodology should include probabilistic reasoning, as some properties cannot be easily formalised by invariants.

Other uses of the generated distributed protocol Event-B model are still being discussed but a software code generation or documentation are of interest. In the following section, we present Event-B communication modelling patterns and the refinement strategy. Then we demonstrate few examples how these patterns were used for developing distributed resource allocation protocol.

IV. MODELLING PATTERNS AND REFINEMENT PLAN

This section mainly describes the developed modelling patterns and their application for modelling the resource allocation protocol. But first, we define a modelling strategy, or in other words refinement plan, which generally could be used as a modelling strategy for any distributed protocol.

A. Refinement strategy

The model development approach we propose is a rather standard and starts with the abstract model which 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 pattern defined in the Subsection IV-C.

The following group of refinement steps 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 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. 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 - the proof is explained in Section V.

The refinement plan could be further extended if distributed protocol developer considers its software implementation. In Iliasov et al. [6] - authors presented a modelling technique to bridge the gap between Event-B formal model and the software protocol implementation. Their approach proposes to introduce an environment into the model and to further decompose communication events to separately model sending and receiving a message. In the following subsections we present how this refinement strategy is modelled with communication patterns and few examples from the resource allocation protocol model.

B. Message context patterns

The Event-B model is made of two key components - machines and contexts which respectively describe dynamic and static parts of the system. The context contains modeller declared constants and associated axioms which can be made visible in machines. In our generic modelling approach we can distinguish two groups of contexts. The first group of contexts (context_abstract) axiomatically define all objects present in the model - carrier set OBJ. For example, objects in the resource allocation protocol are agents and resources, in other protocols they are often called, for example, processes or cohorts. An object is often required to go through a number of protocol steps to achieve its goal. In the context_abstract we also define an enumerated set OBJ_STATUS which would define all program counter values an object can have.

context_abstract

SETS

OBJ, OBJ_STATUS

CONSTANTS

$STATUS_1, STATUS_2 \ldots STATUS_n$ possible object status

AXIOMS

$partition(OBJ\_STATUS, \{STATUS_1\}, \ldots \{STATUS_n\})$

The second group of contexts are communication related and are used to define various messages types. Each message type (e.g. request) in the protocol is defined in a separate context file. In fact latter contexts can be generalised as a pattern and instantiated for the specific message type. To define the context pattern let's first define a generic message set - MSG. In this context we introduce three constant functions for: message source, message destination and (optionally) a message value. Depending on the message type, the source of the message in the distributed resource allocation protocol can be both an agent and a resource. Therefore, in our message pattern we can abstract all communicating objects including agents and resources to a general source - SRC.

Likewise for the message destination constant function form we give a general destination type - DST. In the distributed protocol can messages carry a value (e.g. reply contains promised pointer) so the last generic type we define is VAL and associated constant function (msgv) necessary for extracting value from the message. All constant message functions are surjective ($\twoheadrightarrow$) functions meaning they are total in domain and range. Lastly, each message type contexts contains an axiom which states that there always exists a suitable message between any agent and resource.

context_message_type

SETS

MSG

CONSTANTS

$msgs, msgd, msgv$

AXIOMS

axm_1 $msgs \in MSG \twoheadrightarrow SRC$ message source

axm_2 $msgd \in MSG \twoheadrightarrow DST$ message destination

axm_3 $msgv \in MSG \twoheadrightarrow VAL$ message value

axm_4 $\forall s, d, v \cdot s \in SRC \land d \in DST \land v \in VAL \Rightarrow$
$(\exists m \cdot msgs(m) = s \land msgd(m) = d \land msgv(m) = v)$

C. Communication event patterns

In order to generalise a distributed protocol modelling process we derived communication event and variable patterns which are presented below. The first step in introducing a new message into a machine is extending that machine with a message associated context (e.g. context_request). The following step is creating a three letter variable msg of type inv_1 to represent that messages channel. However, as messages are added and removed from the channel msg we require a second generic local variable of type inv_2 to locally store what messages have been sent.

$\mathbf{inv_1}$	$\mathsf{msg} \subseteq \mathsf{MSG}$	global message channel
$\mathbf{inv_2}$	$\mathsf{msgo} \in \mathsf{OBJ} \rightarrow \mathbb{P}(\mathsf{MSG})$	sent msg messages

After introducing message variables, one needs to create events in the model which send that message. From modelling various distributed protocols we identified two message sending event patterns. The most widely encountered communication situation is responding to received message - reply event type. We define a reply event pattern (object_reply_MSG) where the main principle of this event is to take a message from one channel and create a response message in the different channel. In this pattern we use constant message functions defined in the context files to select the source and destination of the new message. The reply event pattern has two event parameters - messages ms_1 and ms_2 where messages types must be defined according to the protocol. The first guard (grd_1) of this event states that a message ms_1 must have been sent, or in other words, already an element of that channel. Guards $\mathsf{grd}_{1..4}$ select an appropriate reply message ms_2 by using constant functions defined in the message contexts. They state that a new messages destination is the source of received message and that the source of the new message is a destination of received message. The actions of this event create a new message by adding it to the channel variable, remove responded message and save sent receipt locally. Additional guards would be added according to the distributed protocol specifications.

object_reply_MSG		
any		
	$\mathsf{ms}_1, \mathsf{ms}_2$	
when		
grd_1	$\mathsf{ms}_1 \in \mathsf{msg}_1$	received message
grd_2	$\mathsf{ms}_2 \in \mathsf{MSG}_2 \setminus \mathsf{msg}_2$	create a new message
grd_3	$\mathsf{msg}_2\mathsf{d}(\mathsf{ms}_2) = \mathsf{msg}_1\mathsf{s}(\mathsf{ms}_1)$	
grd_4	$\mathsf{msg}_2\mathsf{s}(\mathsf{ms}_2) = \mathsf{msg}_1\mathsf{d}(\mathsf{ms}_1)$	
then		
act_1	$\mathsf{msg}_2 \coloneqq \mathsf{msg}_2 \cup \{\mathsf{ms}_2\}$	send new message
act_2	$\mathsf{msg}_1 \coloneqq \mathsf{msg}_1 \setminus \{\mathsf{ms}_1\}$	remove message
act_3	$\mathsf{msgo}_2(\mathsf{msg}_2\mathsf{s}(\mathsf{ms}_2)) \coloneqq \mathsf{msgo}_2(\mathsf{msg}_2\mathsf{s}(\mathsf{ms}_2)) \setminus \{\mathsf{ms}_2\}$	
end		

Another type of message sending event we can define is an initiating message sending (object_initiating_MSG). The principle of this event is create a new message once the program counter of an object changes to the specific status. This event pattern only has one event parameter ms_1 message which must be not sent yet (grd_1). The second guard states that the program counter (pctn) of the source message must be at some STATUS defined in the abstract_context. The actions of the event add a new message to the message channel msg_1 and save message receipt locally.

object_initiating_MSG		
any		
	ms_1	
when		
grd_1	$\mathsf{ms}_1 \in \mathsf{MSG}_1 \setminus \mathsf{msg}_1$	create a new message
grd_2	$\mathsf{pctn}(\mathsf{msg}_1\mathsf{s}(\mathsf{ms}_1)) = \mathsf{STATUS}$	program counter at
then		specific value
act_1	$\mathsf{msg}_1 \coloneqq \mathsf{msg}_1 \cup \{\mathsf{ms}_1\}$	send new message
act_2	$\mathsf{msgo}_1(\mathsf{msg}_1\mathsf{s}(\mathsf{ms}_1)) \coloneqq \mathsf{msgo}_1(\mathsf{msg}_1\mathsf{s}(\mathsf{ms}_1)) \setminus \{\mathsf{ms}_1\}$	
end		

D. Loop modelling pattern

In a distributed protocol an object is often required to send multiple messages or repeat the process numerous times. For such scenarios we created a two event loop pattern which would often be combined with message sending patterns to model burst message sending. The first event in this pattern is the loop body event (object_STATUS_b) and body events have _b name extensions. The STATUS and object in the event name would be also modified to correspond to a specific event. The loop is triggered, firstly, if agents program counter status is at certain value. The remaining guards are used to selected an appropriate message or further constrain the event. In case of the sending multiple reply messages, guard select_guards, would be instantiated with predicates $\mathsf{grd}_{1..4}$ from object_reply_MSG pattern.

object_STATUS_b		
any		
	ob	
when		
grd_1	$\mathsf{pctn}(\mathsf{ob}) = \mathsf{STATUS}$	
$\mathsf{grd}_{2..n}$	select_guards	guards for selecting message
then		
act_1	send_message	
act_2	remove_message*	
end		

Another event in this pattern is a loop completion event and events have _c name extension. As name suggests this event detects when the iterative process has been completed. The loop_completed guard would be typically predicated on

locally saved message receipt variable msgo. The action of this event simply updates objects program counter to the next value.

object_STATUS_c
any
ob
when
grd_1 pctn(ob) = STATUS
grd_2 loop_completed_guard
then
act_1 pctn(ob) := NEXT_STATUS
end

V. Distributed Protocol Development

In this section we present only few examples on how the presented communication and loop modelling patterns were used in developing the distributed resource allocation protocol. Even though, the concrete (final) refinement step contained 23 events and a similar number of variables, only 3 events did not fit any pattern or were special cases.

A. Abstract distributed resource allocation protocol model

The distributed resource allocation protocol development approach follows a standard Event-B modelling approach where the abstract model summarises the protocol with a centralised view of the system. In short the objective of the distributed protocol is to enable safe resource locking which we abstract as agents capturing and releasing collections of resources - objectives. The abstract model was split in two machines where the first machine simply models agents capturing and releasing objectives. In the extension of the initial abstract model we introduced resources into the model and decomposing two previously introduced events agent_capture and agent_release.

agent_consume_b
any
rs, ag event parameters : agent and resource
when
grd_1 ag ∈ dom(capt)
grd_2 rs ∈ objr(objt(ag)) resource within the objective
grd_3 rs ∉ union(ran(capt)) not yet consumed resource
grd_4 pct0(ag) := CONSUME
then
act_1 capt(ag) := capt(ag) ∪ {rs} capture new resource
end

In this refinement step agents attempt to capture multiple resources in order to fulfil an objective where objective is a collection of resources. We started modelling by introducing new variables for storing captured resources capt and storing agents objective objt. Resources which belong to the objective can be extracted by applying an objective to the constant function objr which was defined in the abstract context. For the iterative capturing and releasing events we applied the loop pattern as depicted below in the model excerpt. The loop body event agent_consume_b is enabled if agents program counter is at the CONSUME state. Furthermore, the resource must be within agent's objective (grd_2) and not captured by any agent (grd_3). The action of this event (act) simply stores a new resource rs to that agent in the capt variable.

agent_consume_c
any
ag
when
grd_1 capt(ag) = objr(objt(ag)) completed its objective
grd_2 pct0(ag) = CONSUME
then
act_1 pct0(ag) := RELEASE update program counter
end

The loop completion event agent_consume_c would be triggered as soon as the objective has been fulfilled and program counter would be updated to new state - RELEASE. Similarly in this refinement we transform agent_release event according to the pattern presented. To show correctness of the extended model we prove an invariant, which states that no agents can have the same resource captured still this system is not deadlock free.

B. Protocol communication modelling with patterns

The abstract model is refined according to the proposed development method by introducing communication aspects of the protocol. The method also suggest to use a backward protocol unfolding modelling style. Therefore, the abstract model was refined by introducing stage_2 of the protocol first which is responsible for locking resources once a distributed lane has been negotiated. The following paragraphs explain how communication pattern were used to add lock message into the model.

To begin with, the machine m_2 is extended with the message type context context_message_lock which defines constant functions of the message. Then, we create variables lck (inv_1) and lcke (inv_2) which will represent a communication message channel and a memory where sent lock message receipts are locally stored. In this protocol it was more convenient to locally save where message have been sent rather than messages itself.

$\mathbf{inv_1}$	lck ⊆ LCK lock message channel
$\mathbf{inv_2}$	lcke ∈ AGT → ℙ(RES) locally stored messages

To model lock message sending we apply a loop together with initiating message modelling patterns to create two new events agent_lock_b and agent_lock_c. In the protocol lock message is of the reply type, however, since preceding protocol

messages are modelled in the next refinement lock message is an initiating message at this stage but in the next refinement is modified to a reply type. The actions of the loop body event send a new lck message firstly if the program counter of an agent is at LOCK phase. Secondly, the new message must not have been sent already $grd_{1,2}$ and destination of the message (resoure) is within agent's objective grd_3.

agent_lock_b
any
lc
when
grd_1 $lc \in LCK \setminus lck$
grd_2 $lckd(lc) \notin lcke(lcks(lc))$ messages not sent yet
grd_3 $lckd(lc) \in objr(objt(lcks(lc)))$ within the objective
grd_4 $pct2(lcks(lc)) = LOCK$
then
act_1 $lck := lck \cup \{lc\}$ send new message
act_2 $lcke(lcks(lc)) := lcke(lcks(lc)) \cup lckd(lc)$
end

The loop completion event, agent_lock_c, detects the end of the loop and updates the program counter. 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 (grd_1). The action of this event simply updates the program counter to the next state.

agent_lock_c
any
ag
when
grd_1 $lcke(ag) = objr(objt(ag))$ all lock messages sent
grd_2 $pct2(ag) = LOCK$
then
act_1 $pct2(ag) := CONFIRMC$ update program counter
end

VI. Distributed Protocol Verification

Formally specifying a system is not sufficient to guarantee a high safety and correctness confidence degree. In the proposed methodology we regard mathematical proofs as essential for ensuring high system assurance. Therefore, our development method relies on adding and proving properties (invariants) in the Event-B model. The literature review and our experience in developing various distributed protocols however show that formalising sufficient properties in many cases might be too challenging, thus, a wider validation tool support is crucial.

Three different verification techniques were used in developing the resource allocation protocol. From the beginning it was decided to use ProB model checker, which is available as a Rodin Platform plug-in, to validate early protocol designs before formalising and proving properties. To use ProB model checker we had to develop additional message contexts, since ProB built-in constraint solver could not compute bounded sets according to the defined axioms. On the other hand, ProB tool accepts Event-B model as an input, so it was not necessary to manually translate the model. In result, the model checker helped to discover subtle protocol deadlock scenarios which could be replayed on the built-in model animator. The model checker allowed to discover and significantly modify the protocol in early design stages without formalising and proving complicated properties. The state explosion was still a big issue even for checking protocol with few agents and resources would take several hours. The following subsections discuss one of the main completed proofs and the use of probabilistic model checker tool for proving protocol stage_1 termination.

A. Correctness of forming distributed lanes

In order to ensure a safe and deadlock free distributed resource reservation we developed a solution based on notion of distributed lane. In the final model refinement step, we had to prove that the system is deadlock free. To prove that distributed system never deadlocks one could write an invariant which asks to prove that there is always at least one event enabled. However, this results in a very complicated proof so instead we reformulated the problem into a simpler one but for that the final model had to be extended with additional only proof related variables.

The cross blocking deadlock occurs when request messages of multiple agents which are interested in common resources are delayed. Indeed, this scenario is not possible for a simple non-distributed queue model since unique request has only a single slot in the queue. By unifying agents requests indexes over distributed resources, distributed lane solution essentially creates a virtual localised queues. In other words, agents which requested common resources and negotiated a distributed lane, can be virtually collapsed into a single non-compact queue like data structure. Thus, one only needs to prove that distributed lanes are correctly formed.

$$\mathbf{inv_{pro}}\ \forall r, n_1, n_2 \cdot r \in R \wedge n_1, n_2 \in dom(his_{wr}(r)) \wedge \\ \wedge\ n_1 < n_2 \Rightarrow his_{wr}(r)(n1) < his_{wr}(r)(n2)$$

Due to limited space we only state that a distributed lane will be correctly formed if resource always replies with unique value $ppt(r_k)$ value. For the proof we created a new *history* variable which stores chronological values of the promise pointer. Every time a promise pointer is updated in the model the history variable stores that value and increments its write value. The invariant now can be expressed (inv_pro) on the history variable which states that for any resource the recorded promised pointer values increases as history variable write pointer increases. In the model two proofs obligations (ignoring well-defined verification conditions) were generated and proved interactively.

B. Argument for renegotiation termination

The progress of the protocol relies on agents ability to negotiate a distribute lane. As negotiating a distributed lane is a probabilistic process in our protocol we need to demonstrate that a desired state will be eventually reached. The probabilistic reasoning in Event-B is extremely complicated and therefore the decision was made to demonstrate negotiation termination outside the Rodin platform. For that purpose we use a well-known symbolic probabilistic model checker - PRISM. In PRISM modelling framework the model is constructed of individual modules which contain local variables and commands. In generic model we used the functionality of modules to captures resources and agents with an individual module. The modularisation allowed us to capture agents concurrency and model extreme scenarios where some agents could be much faster in sending messages than others.

To model the distributed lane renegotiation we developed a generic stage_1 phase model which would be instantiated to model specific scenarios - defined by a number of agents and resources, and also what resources agent would like to capture. The generic model itself could be partitioned into three major parts: global variable declaration, resource modules and agent modules. The two types of global variables in generic model are both associated with agents. One variable is used to store an promised pointer index the agent receives from the resource as well as to store a special request value which is sent if distributed lane is not negotiated. The second variable, agents state variable, was introduced to disable an agent once a distributed lane was negotiated and in turn affect the transition probabilities. Each resource has an attributed resource module which contains local variable and a command. The local variable we need for resource modules is the promised pointer variable (ppt). The module also only requires a single command to represent request and reply message exchanges. Lastly, each agent also has an associated agent module which models part of the agents algorithm, where all reply messages have been received and now a decisions must be made whether to renegotiate or create a distributed lane. Having designed the generic model we developed a simple program which would create random instantiated PRISM models and simulate them until either all distributed lanes were negotiated or a time-out was reached. The experiment was repeated for a number of scenarios and results were plotted for analysis. A number of steps taken to terminate was plotted against the number of resources and agents in that scenario.

VII. Conclusions and Future Work

In this work we described a refinement and proof based methodology for developing high assurance distributed protocols. To facilitate the method we rely on a firmly established formal modelling framework which provides a refinement based development and a number of useful tools. Communication modelling patterns and a generic refinement strategy presented in this paper are essential towards our main research objective of automatically generating refinement chains from high-level protocol descriptions. These technical contributions were mainly defined by developing a variety of distributed protocols including the distributed resource allocation protocol presented here.

In the paper we also discuss the verification challenges of the distributed protocol. Available plug-ins, particularly, the ProB model checker and its Rodin animation plug-in was extremely useful in discovering subtle deadlock scenarios. Still, due to complex protocol behavior a state-space problem was clear as a single exhaustive model checking run would take hours (even for small scenarios). Still, an alternative option to complete proofs at early development phase was not an appealing choice. As some of early deadlock scenarios discovered were intricate and unlikely captured by an invariant or else proving strong invariant would have been even more time consuming. Demonstrating the termination of the distributed lane negotiation was also decided to be completed outside the Event-B environment as existing probability theory support such as [7] would likely have not been sufficient. Even the discussed proof was not completed systematically and it was necessary to further abstract the problem and add additional variables into the model. The main focus of the future work is development of a high-level protocol specification language which would be translated into a Event-B refinement chain.

Acknowledgements. This work is supported by an iCASE studentship (EPSRC and Siemens Rail Automation).

References

[1] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 2013.

[3] R.-J. Back. *Refinement calculus, part ii: Parallel and reactive programs*. In Proceedings on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness, REX workshop, pp. 67–93, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[4] M. Leuschel and M. Butler. *Prob: A model checker for b*. In: FME 2003: Formal Methods, pp. 855–874, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[5] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. *Prism: A tool for automatic verification of probabilistic systems*. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 441–444, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[6] A. Iliasov, L. Laibinis, E. Troubitsyna and A. Romanovsky. *Formal derivation of a distributed program in Event-B*. In Shengchao Qin and Zongyan Qiu, editors, Formal Methods and Software Engineering, pp. 420–436, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[7] S. Hallerstede and T.S. Hoang. *Qualitative probabilistic modelling in Event-B*. In: Integrated Formal Methods, pp. 293–312, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[8] B. Siala, T. Bhiri, J.-P. Bodeveix, M. Filali. *An Event-B Development Process for the Distributed BIP Framework*. In: 18th International Conference on Formal Engineering Methods (ICFEM 2016), 14 November 2016 - 18 November 2016 (Tokyo, Japan).

[9] Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., Sifakis, J. *Rigorous component-based system design using the BIP framework*. In: IEEE Software, vol. 28, no. 3, pp. 41–48, May-June 2011. doi: 10.1109/MS.2011.27

[10] C. Hawblitzel, J. Howell, M. Kapritsos, J.R. Lorch, B. Parno, M.L. Roberts, S. Setty, and B. Zill. *IronFleet: proving practical distributed systems correct*. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15). ACM, New York, NY, USA, 1–17.