Formal Derivation of a Distributed Program in Event B

Alexei Iliasov[1], Linas Laibinis[2], Elena Troubitsyna[2], and Alexander Romanovsky[1]

[1] Newcastle University, UK
[2] Åbo Akademi University, Finland
{alexei.iliasov, alexander.romanovsky}@ncl.ac.uk
{linas.laibinis, elena.troubitsyna}@abo.fi

Abstract. Achieving high dependability of distributed systems remains a major challenge due to complexity arising from concurrency and communication. There are a number of formal approaches to verification of properties of distributed algorithms. However, there is still a lack of methods that enable a transition from a verified formal model of communication to a program that faithfully implements it. In this paper we aim at bridging this gap by proposing a state-based formal approach to correct-by-construction development of distributed programs. In our approach we take a systems view, i.e., formally model not only application but also its environment – the middleware that supports it. We decompose such an integrated specification to obtain the distributed program that should be deployed on the targeted network infrastructure. To illustrate our approach, we present a development of a distributed leader election protocol.

1 Introduction

Development of distributed systems remains one of the more challenging engineering tasks. The complexity caused by concurrency and communication requires sophisticated techniques for designing distributed systems and verifying their correctness. Active research in this area has resulted in a large variety of distributed protocols and approaches for their verification. However, these techniques emphasise the creation of a mathematical model establishing algorithm properties and per se do not provide an unambiguous recipe on how to develop a distributed program that would correctly implement a desired algorithm. Moreover, these techniques often ignore the impact of deploying the developed software on a particular network infrastructure. In this paper we present a complete formal development of a distributed protocol (a fairly common variation of leader election [9]) and demonstrate how state-based modelling and refinement help to alleviate these problems.

The main technique for mastering system complexity is abstraction and decomposition. Our development starts from creating an abstract centralised system specification. In a chain of correctness preserving refinement steps we build

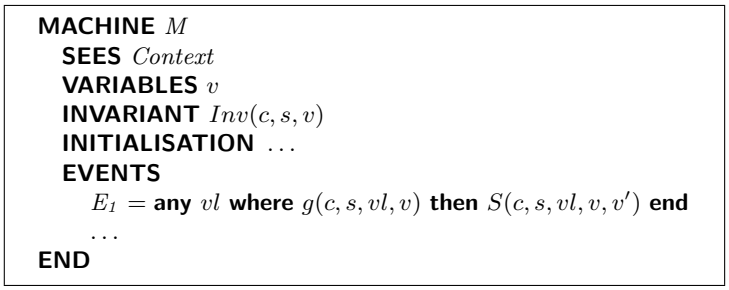

Fig. 1. Structure of Event B model

the details of the distributed algorithm and communication scheme. Our modelling adopts a systems approach: the specification defines not only the behaviour and properties of the algorithm but also the essential assumptions about the network infrastructure. The final step of our refinement chain is decomposition. We rely on the modularisation extension of Event B [2] to decompose the specification into separate modules representing the communicating processes and the middleware. The obtained specification of processes may be further refined into runnable specifications and treated as programs. Alternatively, a translation may be done into a programming language. Our formal development ensures that the resulting distributed program will behave correctly when deployed on the middleware that satisfies the assumptions explicitly stated in the formal model.

2 Background

We start by briefly describing our development framework. The Event B formalism [2, 16] is a specialisation of the B Method [1], a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. Event B enables modelling of event-based (reactive) systems by incorporating the ideas of the Action Systems formalism [4] into the B Method. Event B is actively used within the FP7 ICT project DEPLOY [12] to develop dependable systems from various domains.

2.1 Modelling and Refinement in Event B

The Event B development starts from creating an abstract system specification. The general form of an Event B model is shown in Figure 1. Such a model encapsulates a local state (program variables) and provides operations on the state. The actions (called *events*) are defined by a list of new local variables (parameters) vl, a state predicate g called *event guard*, and a next-state relation S called *substitution* (see the **EVENTS** section in Figure 1). Event parameters and guards may be sometimes absent leading to the respective syntactic shortcuts starting with keywords **when** and **begin**.

The event guard g defines the condition or a set of states when the event is *enabled*. The relation S is expressed as either a deterministic or non-deterministic assignment to the model variables. One form of a non-deterministic assignment

used in the paper is the selection of a value from a set expression, written as $v :\in Set$, where Set is an non-empty set of possible values.

The **INVARIANT** clause contains the properties of the system (expressed as state predicates) that should be preserved during system execution. These define the *safe states* of a system. In order for a model to be consistent, invariant preservation should be formally demonstrated (i.e., proved). Data types s, constants c and relevant axioms are defined in a separate component called *context* (clause **SEES** in Figure 1).

The cornerstone of the Event B method is *refinement* – the process of transforming an abstract specification by gradually introducing implementation details while preserving correctness. It allows us to transition from an abstract, non-deterministic model to a detailed, deterministic program implementing a system. Since Event B models are state-based, data refinement – a technique of reinterpreting a model using different state models – is at the core of most refinement proofs. For a refinement step to be valid, every possible execution of a refined machine must correspond to some execution step of its abstract machine.

The consistency of Event B models, i.e., verification of well-formedness and invariant preservation as well as correctness of refinement steps, should be formally demonstrated by discharging relevant *proof obligations*, which collectively define the *proof semantics* of a model [2]. The Rodin platform [18], a tool supporting Event B, is an integrated environment that automatically generates necessary proof obligations and manages a collection of automated provers that would try to autonomously discharge the generated theorems. At times a user intervention would be necessary to guide the provers or construct a complete proof in an interactive proving environment. The level of automation in proving is high enough to make realistic development practical. On average, the rate of assisted proofs is about 20%, of which only a small percentage goes beyond giving one or two hints to the provers.

2.2 Modelling Modular Systems in Event B

Recently the Event B language and tool support have been extended with a possibility to define modules [11, 17] – components containing groups of callable atomic operations. Modules can have their own (external and internal) state and the invariant properties. The important characteristic of modules is that they can be developed separately and, when needed, composed with the main system.

A module description consists of two parts – *module interface* and *module body*. Let M be a module. A module interface MI is a separate Event B component. It allows the user of module M to invoke its operations and observe the external variables of M without having to inspect the module implementation details. MI consists of external module variables w, constants c, sets s, the external module invariant $M\_Inv(c, s, w)$, and a collection of module operations, characterised by their pre- and postconditions, as shown below.

The primed variables in the operation postcondition stand for the final variable values after operation execution. If some primed variables are absent, this means that the corresponding variables are unchanged by an operation.

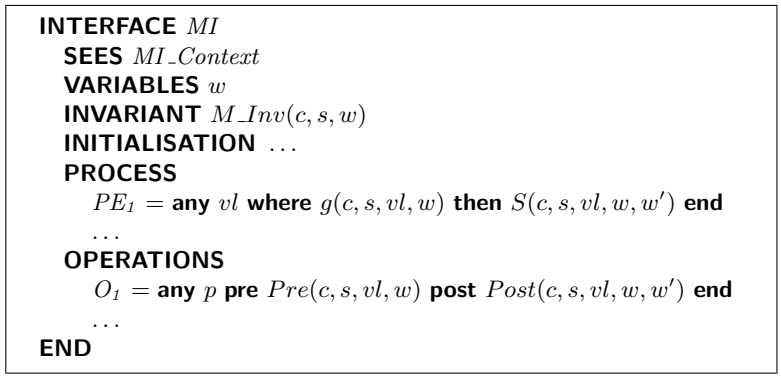

Fig. 2. Interface Component

In addition, a module interface description may contain a group of standard Event B events under the **PROCESS** clause. These events model autonomous module thread of control, expressed in terms of their effect on the external module variables. In other words, the module process describes how the module external variables may change between operation calls.

A module development starts with the design of an interface. Once an interface is defined, it cannot be altered in any manner. This ensures that a module body may be constructed independently from the model relying on the interface of the module. A module body is an Event B machine. It implements the interface by providing the concrete behaviour for each of the interface operations. The interface process specification may be further refined like a normal subset of Event B events. A set of additional proof obligations are generated to guarantee that an operation has a suitable implementation in the implementing machine.

When module M is imported into another Event B machine, the importing machine may invoke the operations of M and access (read) the external (interface) variables of M. To make a specification of a module generic, in $MI\_Context$ we can define some constants and sets (types) as parameters. Their properties then define the constraints to be verified when a module is instantiated.

Next we present a formal development of a distributed system that illustrates the various aspects of modelling and refinement in Event B.

3 Modelling of a Leader Election Protocol

The main goal of this paper is to present an entire formal development of a distributed system in Event B. The system implements a leader election protocol, i.e., its purpose is to elect a single leader among all the participating processes. The solution is inspired by the bully algorithm [9]. In its simple interpretation the algorithm ensures that the process with a largest id wins from all the processes willing to become a leader.

Our development strategy is as follows. The development starts from a trivial high-level specification that "magically" elects the leader. In a number of steps we obtain a model of a centralised leader election algorithm. Then we gradually decentralise this model by refining its data structure and behaviour. Several

refinement steps aim at decoupling the data structures of the individual processes and introducing the required communication mechanism.

In this last, more challenging refinement step, we decompose the specification to separate the model of communication environment (middleware) from the models of constituent processes. To achieve this, we rely on the modularisation extension of Event B. Finally, we show how the process specifications may be converted into runnable code. Due to a space limit we do not present the complete specifications produced at each refinement step. Instead we describe the more interesting aspects of each particular model. The complete Event B development is available at [14].

3.1 Abstract Model of Leader Election

Abstract specification Our development starts with creating an abstract model that defines a single variable *leader* and one-shot leader election abstraction. Assume that the system has n processes and the process ids are from $1 \mathinner{..} n$. The leader election protocol is made of a single event that atomically selects a new leader value:

$$elect = \textbf{any } nl \textbf{ where } nl \in 1 \mathinner{..} n \textbf{ then } leader := nl \textbf{ end}$$

First Refinement In the first refinement step we start to introduce some localisation properties of the algorithm. Each process is able to decide (vote) on whether it wants to become a new leader or not. Such a decision is made by a process independently of other processes and is recorded in a global vector of decisions: $decision : 1 \mathinner{..} n \pfun 0 \mathinner{..} n$.

When a process votes, it puts into the decision vector either its id (process identifier), indicating that it is willing to be a new leader, or 0, indicating the opposite: $\forall i \cdot i \in dom(decision) \Rightarrow decision(i) \in \{0, i\}$.

To determine the leader among the set of willing processes, we compare their "bully" id's. A new leader is a process with the maximal id among the processes that are willing to be leaders. Assuming the decision vector is complete, the new leader is the process with the largest id, i.e., $max(ran(decision))$, where ran is the function range operator.

Unfortunately, all the processes may refuse being a leader and then the election has to be restarted. This means that a protocol round is potentially divergent. To avoid this, we exclude the situation when every process decides not to be a leader. This is achieved by requiring that any process willing to initiate the protocol is also committing to be a leader. The corresponding invariant property states that, whenever the decision vector is not empty, there is a process willing to be a new leader: $card(decision) \geq 1 \Rightarrow max(ran(decision)) \in 1 \mathinner{..} n$.

This invariant guarantees that after a voting round, when the decision vector has records for all the processes, there is a new leader. The protocol may be initiated by any process that has not yet voted (event *initiate*). As soon as a new election is initiated, i.e., $dom(decision) \neq \varnothing$, the remaining processes are free to choose or decline to be a new leader (event *decide*).

$$
\begin{array}{l}
initiate = \textbf{any } idx \textbf{ where} \\
\quad idx \in 1 \mathinner{..} n \\
\quad idx \notin dom(decision) \\
\textbf{then} \\
\quad decision(idx) := idx \\
\textbf{end}
\end{array}
\qquad
\begin{array}{l}
decide = \textbf{any } idx, d \textbf{ where} \\
\quad idx \in 1 \mathinner{..} n \\
\quad idx \notin dom(decision) \\
\quad d \in \{idx, 0\} \\
\quad decision \neq \varnothing \\
\textbf{then} \\
\quad decision(idx) := d \\
\textbf{end}
\end{array}
$$

Even after the protocol has been initiated, the processes can still continue to "initiate" the election. This effectively corresponds to expressing willingness to become a new leader. The abstract event elect is now refined by the following a deterministic event, computing the new leader id from the vector of process decisions.

$$elect = \textbf{when } dom(decision) = 1 \mathinner{..} n \textbf{ then } leader := max(ran(decision)) \textbf{ end}$$

3.2 Decentralising Leader Election

Second refinement After the first refinement the leader election is modelled in a centralised way – processes are able to access the global decision vector *decision*. It yields a simple model but prevents a distributed implementation. Our next refinement step aims at decentralising the model (and thus the localisation of model state). For each process i, we introduce $other(i)$ – a local, process-specific version of the global decision vector: $other \in 1 \mathinner{..} n \rightarrow \mathbb{P}(0 \mathinner{..} n)$.

Based on the information contained in $other(i)$, the process i should be able compute an overall leader without consulting the global vector *decision*. In the invariant we postulate that a local knowledge of a process is a part of the global decision vector: $\forall i \cdot i \in 1 \mathinner{..} n \Rightarrow other(i) \subseteq ran(\{i\} \lhd decision)$, where $\lhd$ is the domain subtraction operator. At the same time, $other(i)$ does not include the process decision stored separately.

To populate their local versions of the decision vector, the processes have to communicate between each other. Once the process has voted, it starts communicating its vote to other processes. Symmetrically, a process populates its local knowledge by receiving messages from the other processes. As a simple model of communication, for each process i, we introduce the set $recv(i)$ – a set of the processes from which it has received their vote messages, and the set $pending(i)$ – a set of the processes that it has committed to communicate its decision to. A process must communicate its decision to all other processes.

We do not consider here process and communication failures. Process crashes and message loss do not affect the correctness properties but make it impossible to demonstrate the progress (convergence) of the protocol (unless, of course, there is an upper bound on the number of process and message failures). We prefer to produce a stronger, convergent model first and then consider a case where individual communications steps cannot be proven convergent due to potentially infinite retransmission attempts recovering from lost messages. Our intention is to deal with process and communication failures in the model of the middleware introduced after the decomposition step.

In the following invariant we define the properties of our communication scheme. The messages committed by one process to be sent are not yet received by another process:

$$\forall i, j \cdot i \in 1 \mathinner{..} n \wedge j \in 1 \mathinner{..} n \Rightarrow (i \in pending(j) \Rightarrow j \notin recv(i))$$

Moreover, for each process, the local version of the decision vector $other(i)$ is an exact slice of the global decision vector, formed from the received messages:

$$\forall i \cdot i \in 1 \mathinner{..} n \Rightarrow decision[recv(i)] = other(i)$$

where [...] is the relational image operator.

The behavioural part of the communication model comprises two new events: send and receive. The event send models sending a decision message to some destination process to. The event is potentially divergent. It will be made convergent in the next refinement step. The event receive models the reception of a decision message. It uses the message to update the local knowledge ($other(to)$) of a process as well as the sets of the received and pending messages.

$send =$	$receive =$
any idx, to **where**	**any** idx, to **where**
$idx \in dom(decision)$	$idx \in dom(pending)$
$to \in 1 \mathinner{..} n \setminus \{idx\}$	$to \in pending(idx)$
$idx \notin recv(to)$	**then**
then	$recv(to) := recv(to) \cup \{idx\}$
$pending(idx) :=$	$other(to) := other(to) \cup \{decision(idx)\}$
$pending(idx) \cup \{to\}$	$pending(idx) := pending(idx) \setminus \{to\}$
end	**end**

The purpose of the decentralisation performed at this refinement step is to ensure that, once the local decision vector is completely populated, a process can independently elect the leader. To represent the locally selected leaders, we split (data refine) the abstract variable $leader$ into a vector of leaders, one for each process. We define the gluing invariant that connects the new vector $leaders$ with the abstract variable $leaders \in 1 \mathinner{..} n \nrightarrow 1 \mathinner{..} n$ such that $\forall i \cdot i \in dom(leaders) \Rightarrow leaders(i) = leader$.

The central property of this refinement step is that the leader id is determined from the local knowledge is the actual leader defined by the global decision vector:

$$\forall i \cdot i \in 1 \mathinner{..} n \ \wedge \ recv(i) = dom(\{i\} \lhd decision) \ \Rightarrow \\ max(ran(decision)) = max(other(i) \cup \{decision(i)\})$$

This is an essential property with respect to the protocol correctness. It also justifies refining the event *elect* into its decentralised version, where each process is able to compute the common leader once *other(i)* contains the decisions of all other processes: $leaders(idx) := max(other(idx) \cup \{decision(idx)\})$.

3.3 Refining Inter-Process Communication

Third refinement Our next refinement steps aims at achieving further decoupling of process data structures. Currently, to send a decision message, a process should access the $recv$ variable of the targeted recipient to check that it has not

received this message. Such an access to the process local data can be avoided if for each process i we introduce a history of the recipients of messages sent by i – $sent(i)$, where $sent \in 1\,..\,n \to \mathbb{P}(1\,..\,n)$. Correspondingly, event **send** is refined as follows

$send =$ **any** idx, to **where**
$\quad \ldots$
$\quad \underline{to \notin sent(idx)}$ // **instead of** $idx \notin recv(to)$
then
$\quad sent(idx) := sent(idx) \cup \{to\}$
$\quad \ldots$
end

Since *sent(i)* includes all the messages currently being transmitted, the pending messages constitute the subset of the outgoing messages history: $\forall i \cdot i \in 1\,..\,n \Rightarrow pending(i) \subseteq sent(i)$. Now we can formulate the following property central to the model of communication mechanism.

$$\boxed{\forall i, j \cdot i \in 1\,..\,n \Rightarrow (j \in sent(i) \setminus pending(i) \Leftrightarrow i \in recv(j))}$$

It postulates that if a process j has received a message from a process i then the process i has sent the message to j and this message is not currently in transition (not in the set *pending*). The same property holds in the other direction: what has been sent and is not being transmitted has been received. This allows us to conclude that the combination of *sent*, *pending* and *recv* describes a *one-to-one asynchronous communication channel*.

Fourth refinement As a result of introducing the message history *sent*, the vector *recv* becomes redundant. Therefore, it may be data refined by a simpler data structure, *irec*, storing only the number of received messages. This simplifies the model, e.g., by allowing to replace adding elements to a set by incrementing the message count *irec*, where $irec \in 1\,..\,n \to 0\,..\,(n-1)$ and $\forall i \cdot i \in 1\,..\,n \Rightarrow irec(i) = card(recv(i))$.

Fifth Refinement The final step towards achieving the localisation of process data is to avoid direct access to the memory of another agent[3]. One remaining case is an action of the *receive* event updating variable $other(i)$ (see the complete event *receive* definition above): $other(to) := other(to) \cup \underline{\{decision(idx)\}}$.

Our solution is to communicate the decision of the sending process along with the id of the destination process. The value $decision(idx)$ becomes embedded into a message sent by the process idx to the process to. We refine the history of sent messages *sent* by an extended version *xsent*: $xsent \in 1\,..\,n \to (1\,..\,n \nrightarrow 0\,..\,n)$ such that $\forall i \cdot i \in 1\,..\,n \Rightarrow sent(i) = prj1[xsent(i)]$. A similar relation is defined for $pending(i)$. The decision of the sender is contained in the second projection: $\forall i \cdot i \in 1\,..\,n \wedge xsent(i) \neq \varnothing \Rightarrow \{decision(i)\} = prj2[xsent(i)]$.

Let us now analyse the information presented in *xsent*. The domain of *xsent* is the name of a sending process, while its range is a set of pairs of the form of

[3] The sole exception we are going to allow in our model concerns the values of $pending(i)$, which may be read by another process j. Around this exception we are going to build a model of inter-process communication.

(*target process*, *process decision*). *xsent* is a set of triplet that have the structure of a simple network protocol message:

$$\boxed{\langle source\_address \rangle \mid \langle target\_address \rangle \mid \langle payload \rangle}$$

The address fields are process names and the decision of a sending process. The result of this refinement step is a model that reflects the essential features of a distributed system. Namely, it separates the private and externally visible memory of the processes and explicitly defines inter-process communication. The communication is based on message passing in a point-to-point network protocol.

4 Deriving Distributed Implementation

Before presenting the decomposition of the model of leader election, we first describe the general structure of a distributed system that we aim at.

Our goal is to implement a distributed software that will operate on top of the existing hardware and some network infrastructure (*middleware*). We assume that middleware is a generic platform component and its sole functionality is to deliver messages between processes. In our model we make the following assumptions about the middleware behaviour:

- the middleware implements a simple point-to-point communication protocol; as a message it expects a data structure containing source and target addresses of the network points as well as the data to be delivered to the target point;
- for any message sent, it is guaranteed that the message is eventually delivered[4];
- when a message is delivered, the sender gets a delivery receipt;
- the middleware is not able to access the internal memory of a process; it only observes the buffer of output messages.

The description fits any packet-oriented protocol that has the capability of acknowledging the reception of a message, i.e., TCP/IP. While modelling a distributed system, we consider the communications among the processes as observations of the messages sent and received by the processes. The communication history of an individual process is represented by two message sequences – for sent and received messages respectively.

We define the following data structures:

$il \in \mathbb{N}$	index of the last message in input history
$ol \in \mathbb{N}$	index of the last message in otput history
$ih \in 1 \ldots il \to MSG$	input history sequence
$oh \in 1 \ldots ol \to MSG$	output history sequence
$r \in \mathbb{N}$	index of the last sent message in input history

where MSG is an type of process messages. The current queue of outgoing messages is then a particular slice of the output history:

[4] In reality, this means that the failure to deliver a message aborts the whole protocol. The consequences of this are outside of the scope of this paper.

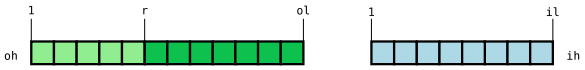

Fig. 3. Output history (oh), output queue (the darker part of oh) and input history (ih).

$(r+1\,..\,ol \lhd oh)$	output queue
$ol - r \leq L$	output queue length constraint

where L is the maximum length of the output queue (that is, the sender buffer size). At any given point of time the output message history consists of the messages already sent and the messages produced but not yet delivered. The variable r points at the last message that has been reported as delivered by the environment. When an environment successfully delivers a message, the variable r is incremented. The out-of-order message delivery is possible but for simplicity we focus on a simpler, ordered delivery (the protocol itself is insensitive to out-of-order delivery). The oldest message awaiting delivery is located at index $r+1$ in the output history. Consequently, the restriction of the output history $r+1\,..\,ol \lhd oh$ gives us a sequence of all the messages awaiting delivery. The middleware constantly observes the changes made to the output histories of processes and reacts on the appearance of a new message awaiting delivery[5].

To model process communication, we define callable operations for each process. These operations will be invoked by the middleware every time a message is delivered or received. We specify these operations in the style described in subsection 2.2. Each process has two such operations, receive_msg and deliver_msg:

$$
\begin{array}{l}
receive\_msg = \textbf{any } m \textbf{ pre} \\
\qquad\qquad m \in MSG \\
\qquad\quad \textbf{post} \\
\qquad\qquad ih' = ih \cup \{il+1 \mapsto m\} \wedge \\
\qquad\qquad il' = il + 1 \\
\qquad\quad \textbf{end} \\
deliver\_msg = \textbf{pre } ol > r \textbf{ post } r' = r + 1 \textbf{ end}
\end{array}
$$

The operation receive_msg is invoked when a new message is delivered to a process. The operation saves the message into the input history. An invocation of deliver_msg by the middleware informs a process about the delivery of the oldest message in the output queue.

Let P, Q be processes and P_pid, Q_pid be their ids (names). The process communication is modelled via the operations P_receive_msg and Q_deliver_msg that correspond respectively to the P and Q instances of receive_msg and deliver_msg. The asynchronous communication between P and Q is specified by two symmetric middleware events Q_to_P and P_to_Q, where

[5] Event B does not have a notion of fairness so it is possible that the middleware delays the delivery of a message for as long as there are other messages to deliver. We have proven that this does not affect the protocol progress.

$Q\_to\_P =$ **any** msg **where**
$msg \in 0..n$
$Q\_ol > Q\_r$ //process has an undelivered message
$P\_pid \mapsto msg = Q\_oh(Q\_r + 1)$
then
$nil :=$ P_receive_msg(msg) // the message is delivered to P
$nil :=$ Q_deliver_msg // and removed from the queue of Q
end

Here *nil* is a helper variable to save a void result of a operation call. The event P_to_Q is defined similarly. Both Q_to_P and P_to_Q are refinements of the abstract event receive.

Above we considered the case with two processes, P and Q. However, the presented approach can be generalised to any number of processes[6].

4.1 System Architecture

The goal of the planned decomposition refinement step is to derive the distributed architecture that supports the communication scheme described above. The general representation of the system architecture after decomposition (in a simple case with only two processes) is shown in Figure 4. The abstract, monolithic model is refined by a model representing the communication middleware, which references the involved processes realised as separate modules. The middleware accesses the modules via the provided generic interface (hence, all the processes are based on the single interface). The interface of a module includes not only the operations receive_msg and deliver_msg but also the specification of an autonomous process thread of control (or, simply, a process thread).

In this part of an interface we define the effect of a process thread on the interface variables. In a distributed system, the specification of a process thread should at least define how and when the process adds messages to the output queue. We also require that a process does not change the input history and the part of output history preceding the output queue. Also, the output message queue may only be extended by a process thread (that is, the middleware would not create messages on its own). These conditions are mechanically generated as additional proof obligations for a model.

A process thread is defined via a number of events. At the decomposition level, these events define how the process thread may update the interface variables. Since the interface variables are used to replace the abstract model variables, it is necessary to link the interface events with the abstracts events of the model before decomposition. Such a link is defined at the point of module import (in the middleware model). It is necessary to specify which interface events are used to refine abstract model events. There are two typical scenarios to achieve this: (i) distributing the abstract events among the interface events, or

[6] At the moment one has to fix the number of processes at a decomposition step. We are working on improvements to Rodin that would allow us to model generic decomposition steps and later instantiate them with any number of processes.

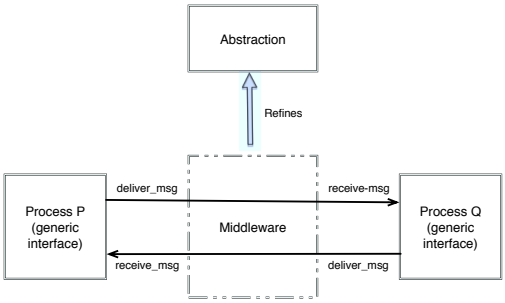

Fig. 4. Decomposition architecture

(ii) splitting the same abstract event into several interface events. In our model it is the latter case as we move the behaviour of a process into its own module.

Next we present how this general decomposition strategy can be applied to derive a distributed implementation of the leader election protocol.

4.2 Decomposition of the Leader Election Model

The essence of the decomposition refinement step is the data refinement of the various data structures of the leader election protocol into input and output message histories of the individual processes. For the sake of simplicity, the election is done among two process called P and Q, which differ, at the interface level, only by their process id.

The overall scheme of the accomplished decomposition refinement is shown in Figure 5. The result of the fifth refinement, elect5, is refined by the middleware model elect6. This model imports two instances of the generic module interface $Node$. All the variables and most of operations of the interface corresponding to processes P and Q appear with the prefixes $P\_$ and $Q\_$ respectively. This is a feature of the modularisation extension that helps to avoid name clashes when importing several interfaces. The variables of the abstract model elect5 are split into two groups. The variables *decision* and *leaders* become internal, "phantom" variables of the process instances, i.e., they cannot be accessed by middleware. These variables are only used in the process thread. The other variable group including *other*, *irec*, *xsent*, and *xpending* is replaced (data refined) by the corresponding input and output histories of the process instances.

The number of message received from other process is related to the length of the input history of a process, while the local knowledge of a process is same as the information contained in its input history. The following are the data refinement conditions for these two variables (for the case of process P)

$$irec(P\_pid) = P\_il$$
$$other(P\_pid) = ran(P\_ih)$$

The process output history, oh, is a suitable replacement for $xsent$. The abstract notion of not yet delivered messages ($xpending$) maps into the output message queue. The refinement relation for $xsent$ and $xpending$ is the following:

$$xsent(P\_pid) = ran(P\_oh)$$
$$xpending(P\_pid) = ran(P\_r + 1 \mathinner{..} P\_ol \lhd P\_oh)$$

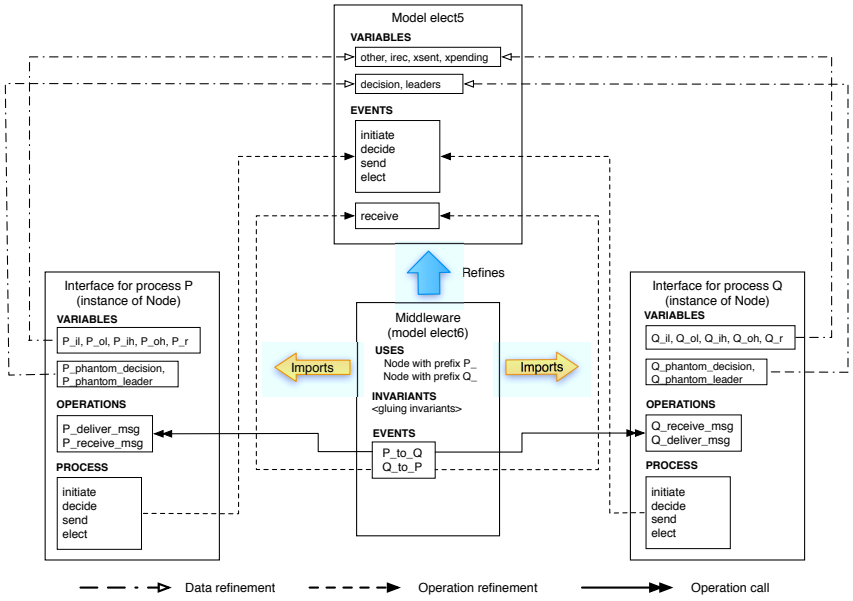

Fig. 5. The structure of the decomposition refinement

From earlier refinement steps we know that $xpending(P\_pid) \subseteq xsent(P\_pid)$. This coincides with the definition of the output queue as a subset of the output history. Such simple mapping between message histories and the variables of the previous (centralised) model is possible because they describe the same idea – asynchronous communication – albeit in differing terms.

The events of the abstract model are also split into two groups. The event receive is now refined by the middleware events implementing the point-to-point process communication, P_to_Q and Q_to_P. The remaining events, initiate, decide, send, and elect, become a part of the process thread of each process. Therefore, these events are also distributed, each dedicated to the functionality of a single process.

4.3 Towards runnable code

The decomposition step leaves us with a specification of a process separated from the specification of the middleware. To implement the model as a software system, we have to achieve the following goals: (i) ascertain that the middleware model is compatible with the deployment infrastructure; (ii) convert the process specification into process software. The former task may be approached by using the middleware model as a blueprint to be matched against the existing middleware implementation, e.g, via model-based testing. A logically simpler, though normally impractical, alternative is to pursue the top-down approach and refine the middleware model all the way to machine code or even hardware description. For the latter task, one normally employs a code generator, which essentially accomplishes a large-scale unproven refinement step relating two formal languages:

a target programming language and our modelling language, Event B. Unfortunately, no such tool is yet available for Event B. Nevertheless, we have conducted a manual, line-by-line translation into Java [15]. Let us note that, whichever approach we take, we have to make a transition from a completely proven artefact to a system with an unproven, informal part. One way to reduce the gap between a model and its runnable version is to treat a deterministic model as a program. For this, it suffices to build an Event B model interpreter, which, in principle, can be achieved by first defining an Event B operational semantics.

Let v be the variables, E the set of events and Σ_0 the set of initial states of an Event B model. Also, let the set of model states be defined as $\Sigma = \{v \mid I(v)\}$, where $I(v)$ is the model invariant and $\Sigma_0 \subseteq \Sigma$. A distinguished value nil $\notin \Sigma$ denotes the state of an uninitialised model. An event $e \in E$ is interpreted as a next-state relation $e : \Sigma \leftrightarrow \Sigma$. The behaviour of the model is then given by a transition system $(\Sigma \cup \{\text{nil}\}, \rightarrow)$ where $\rightarrow$ is defined by the following rules:

$$\frac{v \in \Sigma_0}{(\text{nil}) \rightarrow (v)} \qquad \frac{v \mapsto v' \in \bigcup E}{(v) \rightarrow (v')}$$

The first rule initialises the model. The second one describes state transitions defined by a combination of all the model events. It may be proven that the proof semantics of Event B implies that, unless a deadlock is reached, there is always a next state v' and it is a safe state. A tool for this style of code generation is available for Event B [6]. Rather than simply interpreting models, it produces equivalent C code.

To obtain a complete operational code, it is necessary to do an 'assembly' step linking the program of a process with the program of the existing networking infrastructure. At the moment this step is completely manual.

4.4 Proof Statistics

The proof statistics (in terms of generated proof obligations) for the presented Event B development is shown below. The numbers here represent the total number of proof obligations, the number of automatically and manually proved ones, and the percentage of manual effort. Here the models elect0-elect6 are the protocol refinement steps. IProcess is the process module interface component. QuickPrg is one specific realisation of the process module body that may learn the overall leader before the completion of the protocol. Other process implementations (found in [14]) do not incur any proof obligations. Two models stand out in terms of proof effort: elect2 is an important step giving the proof of the protocol correctness, while elect6 accomplishes model decomposition. About 35% of total manual proofs are related to the decomposition step.

Step	Total	Auto	Manual	Manual %
elect0	3	3	0	0%
elect1	16	13	3	19%
elect2	76	61	15	20%
elect3	23	23	0	0%
elect4	19	12	7	37%

Step	Total	Auto	Manual	Manual %
elect5	37	29	8	22%
elect6	131	106	25	19%
IProcess	53	44	9	17%
QuickPrg	19	16	3	16%
Overall	377	307	70	19%

5 Conclusions

In this paper we have presented an application of Event B and its modularisation extension to the derivation of a correct-by-construction distributed program. The formal model construction is a top-down refinement chain, based on the ideas pioneered in Dijkstra's work on program derivation [7]. We start with a centralised and trivially correct model and end up with a localised, deterministic model of a process. Small abstraction gaps between models constructed at each refinement step have simplified protocol verification and allowed us to better manage the complexity traditionally associated with distributed programs.

In the domain of refinement-driven development, the closest work is the development of distributed reference counting algorithm [5] and also topology discovery model [10]. Both present verification of fairly intricate protocols and propose developments starting with a simple, centralised abstraction that is gradually replaced by a complex of communicating entities. The focus of these works is on the investigation of the consistency properties of the relevant protocols. In contrast, we place an emphasis on the role of refinement as an engineering technique to obtain correct-by-construction software.

Another closely related work is derivation of a distributed protocol by refinement in [3]. The development finishes with a formal model of a program to be run by each process. Our approach has a number of significant distinctions. Firstly, we perform the actual decomposition and separate processes from middleware. Secondly, we explicitly define our assumptions about the middleware providing the correct infrastructure for deploying the derived distributed program.

Walden [19] has investigated formal derivation of distributed algorithms within the Action Systems formalism [4], heavily relying on the superposition refinement technique. These ideas, while not supported by an integrated toolkit, are very relevant in the domain of Event B.

Butler et al. [13, 8] have presented a number of formal developments addressing refinement of distributed systems. The presented work focuses on Event B modelling of a range of two-process communication primitives.

There are a number of model checking approaches to verifying distributed protocols. Model checking is a technique that verifies whether the protocol preserves certain properties (e.g., deadlock freeness) by fully exploring its (abridged) state space. Refinement approaches take a complementary view – instead of verifying a model extracted from an existing program, they promote a derivation of a program with in-built properties. The correct-by-construction development allows us to increase the complexity of a formal model gradually. This facilitates comprehension of the algorithm and simplifies reasoning about its properties.

The main contribution of this paper is a complete formal development of a distributed program implementing a leader election protocol. The formal development has resulted in creating not only a mathematical model of distributed software but also its implementation. The systems approach that we have adopted has allowed us not only create the programs to be run by network processes but also explicitly state the assumptions about the middleware behaviour. These assumptions allow the designers to choose the appropriate type of middleware

on which the derived program should be deployed. We have conducted a code generation experiment and ascertained that the constructed program behaves as expected [15]. With a tool support, obtaining runnable code would be straightforward. As a continuation of this work we plan to model and derive the code of a fully operational distributed control system.

Acknowledgments

This work is supported by the FP7 ICT DEPLOY Project and the EPSRC/UK TrAmS platform grant.

References

1. Abrial, J.R.: The B-Book. Cambridge University Press (1996)
2. Abrial, J.R.: Modelling in Event-B. Cambridge University Press (2010)
3. Abrial, J.R., Cansell, D., Mery, D.: A mechanically proved and incremental development of IEEE 1394. Formal Aspects of Computing 14, 215–227 (2003)
4. Back, R., Sere, K.: Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8(3), pp.1-23 (1996)
5. Cansell, D., Méry, D.: Formal and incremental construction of distributed algorithms: on the distributed reference counting algorithm. Theoretical Computer Science, Elsevier 364, 318–337 (November 2006)
6. Degerlund, F., Walden, M., Sere, K.: Implementation issues concerning the action systems formalism. In: Proceedings of the Eighth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT'07). IEEE Computer Society (2007)
7. Dijkstra, E.: A Discipline of Programming. Prentice-Hall International (1976)
8. Fathabadi, S., Butler, M.: Applying Event B Atomicity Decomposition to a Multi Media Protocol. In: *Proceedings of FMCO* (2010)
9. Garcia-Molina, H.: Elections in distributed computing systems. IEEE Transactions on Computers 31(1) (1982)
10. Hoang, T., Kuruma, H., Basin, D., Abrial, J.R.: Developing topology discovery in Event B. Science of Computer Programming, Elsevier 74 (November 2009)
11. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D., Latvala, T.: Supporting Reuse in Event B Development: Modularisation Approach. In *Proceedings of Abstract State Machines, Alloy, B, and Z (ABZ 2010), Lecture Notes in Computer Science*, Vol.5977, pp. 174-188, Springer, 2010
12. IST FP7 project DEPLOY: Online at http://www.deploy-project.eu/
13. M.Butler, Yadav, D.: An incremental development of the Mondex system in Event B. Formal Aspects of Computing 20, 61–77 (2008)
14. Protocol, L.E.: Event B specification (2011), available at http://iliasov.org/modplugin/leaderel2commented.zip
15. Protocol, L.E.: Java implementation (2011), available at http://iliasov.org/modplugin/leaderel_program.zip
16. Rigorous Open Development Environment for Complex Systems (RODIN): Deliverable D7, Event B Language, online at http://rodin.cs.ncl.ac.uk/
17. RODIN modularisation plug-in: Documentation at http://wiki.event-b.org/index.php/Modularisation_Plug-in
18. The RODIN platform: Online at http://rodin-b-sharp.sourceforge.net/
19. Walden, M.: Formal Reasoning About Distributed Algorithms. Åbo Akademi University, Finland (1998), ph.D. Thesis