A distributed design of a network recovery algorithm

Maryam Kamali*, Linas Laibinis, Luigia Petre and Kaisa Sere

Department of Information Technologies,
Åbo Akademi University,
Joukahainenkatu 3–5 A, 20520 Turku, Finland
E-mail: maryam.kamali@abo.fi
E-mail: linas.laibinis@abo.fi
E-mail: luigia.petre@abo.fi
E-mail: kaisa.sere@abo.fi
*Corresponding author

Abstract: The increase in design complexity emphasises the relevance of formal verification techniques for both software and hardware. Formal methods with their mathematical-based modelling can provide proofs of various properties for the designs, thus ensuring a certain degree of complexity control and enhancing the system confidence. There are numerous formal modelling and verification techniques employed in designing complex systems. Typically, they either prove or disprove the correctness of the particular specifications of a system’s algorithms with respect to certain initial requirements. The Event-B formal method has been recently extended to address the gap between specification and implementation, via the so-called modularisation extension. In this paper, we present a modularisation-based derivation of a distributed design for a network recovery algorithm, based on the refinement technique of Event-B. We thus contribute to enhancing the reliability and availability of network designs.

Keywords: wireless sensor-actor networks; WSANs; network recovery algorithm; distributed design; object-orientation; formal method; Event-B; refinement; modularisation; Rodin-tool.


Biographical notes: Maryam Kamali received her MSc in Information Technologies from the Iran University of Science and Technology since 2008. From 2009, she has been working in the Distributed Systems Laboratory of the Turku Centre for Computer Science (TUCS) Graduate School where she is a PhD student, enrolled at Åbo Akademi University. Her research interests include applying formal methods for modelling and verification of communication designs in wireless sensor network and network-on-chip.

Linas Laibinis is a Senior Researcher at Åbo Akademi University and holds a docent degree in Computer Science since 2011. He received his PhD in Computer Science in 2000 on mechanised formal reasoning about computer programmes. His research interests include interactive environments for proof
and program construction, as well as application of formal methods to modelling and development of fault tolerant and/or communicating software systems. He has published about 60 refereed articles.

Luigia Petre is a Senior Lecturer at Åbo Akademi University and holds a docent degree (the equivalent of Habilitation) in Computer Science since 2012. She received her PhD in Computer Science in 2005 on modelling techniques in formal methods. Her research interests include formal methods and their integration, energy-aware computation, network availability, network-on-chip architectures, wireless sensor-actor networks, multi-core systems, time and space dependent computing and meta-modelling. She has about 40 publications.

Kaisa Sere is a Professor in Computer Science and Engineering at Åbo Akademi University since 1997. She received her PhD in 1990 on the formal design of parallel algorithms. She is the founder and leader of the Distributed Systems Laboratory at the Department of Information Technologies at the same university. Her current research interests are within the design of distributed systems, especially refinement-based approaches to the construction of systems ranging from pure software to hardware and digital circuits. She was the leader of the Åbo Akademi University work in the EU IST projects Matisse, Rodin and Deploy. She has supervised 17 PhD theses and currently (co-)supervising seven PhD students. She has published more than 100 refereed articles. She was the General Chair of the international conference Formal Methods 2008, the flagship conference in the field of formal methods.

This paper is a revised and expanded version of a paper entitled ‘Reconstructing coordination links in sensor-actor networks’ presented at Nodic Workshop on Dependability and Security (NODES), Copenhagen, Denmark, 22–23 April 2010.

1 Introduction

Our society relies on software-intensive systems or, differently formulated in Booch (2007) “software lives in the interstitial spaces of our society”. Networked systems are one example of the software omnipresence in the everyday life. The financial systems, including banking and stock exchanges, the travelling systems, including booking flights and hotels, the electric grids administration, the nuclear plants processes, etc., are all examples of essential components of the society being gradually adapted to functioning on-line, via the internet or other networking configurations.

Given the widespread and relevance of software-intensive systems, it is imperative to be able to rely on software, i.e., to be certain of its various features and properties. Formal methods, with their mathematical proving core, are an important instrument in ensuring the integrity of software-intensive systems. Traditionally characterised as hard
to use, due to the requested mathematical background and the lack of automatic tools, nowadays formal methods have matured, to the point where they are considered in industry when developing software-intensive systems (Woodcock et al., 2009). Examples of the industrial undertaking of formal methods are increasing. The famous line 14 of the driver-less Parisian metro (Gerhart et al., 1994), developed in 1998 using the B-method (Abrial, 1996), is the first notable example of a formal method-based development, reviewed in Lecomte (2009). The method used by Siemens for developing the software controlling the line 14 train ensured its correctness in a mathematical manner that effectively eliminated the unit testing from the software life-cycle. Few human resources are now needed to operate the trains and in addition, the trains are faster, hence fewer are needed in total.

More recent examples of the Event-B (Abrial, 2010) formal method usage in industry can be seen for instance with space systems (Fathabadi et al., 2011) and SAP (Jeremy and Wei, 2010). In Event-B, the development of a model is carried out step by step from an abstract to more concrete specifications. Using the refinement approach, a system can be described at different levels of abstraction, and the consistency in and between levels can be proved mathematically. An ideal scenario of formal methods supporting the development of reliable software-intensive systems consists in getting a formal proof that the final software is a correct implementation of the specification. For Event-B, this means in practice that there is a series of intermediate formal models between the specification and implementation, all proved correct together with their derivation from each other. We are now moving one step closer in Event-B towards the ideal scenario, in that we can verify that our derived design is correctly distributed. Consider the example of a network recovery algorithm, where some feature requires the addition of a link among two nodes. We typically specify this feature in Event-B and model its correctness and properties in a centralised manner, for the network as a whole. However, at the implementation phase, this feature has to involve the two network nodes that synchronise, so that each adds the required reference to the other. We need a methodology for transforming our centralised modelled feature of adding a link into a distributed addition of a link among the two nodes. Such a methodology has been recently proposed and is referred to as the modularisation extension (Iliasov et al., 2010) of Event-B.

In this paper, we start from a previously developed (Kamali et al., 2008) and verified (Kamali et al., 2010) recovery algorithm for wireless sensor-actor networks (WSANs) and present the formal derivation of a distributed design for this algorithm. This derivation is highly significant as a contribution to provide reliable and distributed software specifications. Our derived distributed design for the network recovery algorithm has a visible object-orientation style, thus bringing along two more features. First, it supports modifiability and reusability and second, it facilitates an easier translation to object-oriented code. Our work also addresses the availability research, because we formalise the distribution of a fault removal algorithm for network recovery.

Event-B (Abrial, 2001, 2007, 2010) is an extension of the B formalism (Abrial, 1996) for specifying distributed and reactive systems. A system model is gradually specified on several levels of abstraction, always ensuring that a more concrete model is a correct implementation of an abstract model. The language and proof theory of Event-B are based on logic and set theory. The correctness of the stepwise construction of formal models is ensured by discharging a set of proof obligations; if these obligations hold, then the development is mathematically shown to be correct. Event-B
comes with the associated tool Rodin (Abrial, 2007; Rodin Tool Platform, 2006), which automatically discharges part of the proof obligations and also provides the means for the user to discharge interactively the remaining proofs.

We proceed as follows. In Section 2 we shortly overview Event-B, the modularisation extension and the network recovery algorithm. In Section 3 we describe the development of the verified algorithm to the extent needed in this paper. In Section 4 we introduce and discuss our distributed design. We bring forward the impact of our modelling and its advantages in Section 5 and conclude in Section 6, also reviewing related work.

2 Preliminaries

In this section, we briefly overview Event-B, the modularisation extension, and the network recovery algorithm to distribute.

2.1 Event-B

A typical Event-B model consists of two components, called context and machine. The context describes the static part of the model, i.e., it introduces carrier sets $s$ and constants $c$. The properties of these are described as a list of axioms. The machine describes the dynamic part of the model, consisting of a list of variables $v$ and a list of events $E$. The set of values of the variables forms the state of the model. The properties that should be preserved during the execution are formulated as a list of invariant predicates over the state of the model.

An event, modelling state changes, is formed of a guard $G$ and a substitution $S$; the latter describes the next-state relation between the variables values $v$, before the occurrence of the event and the variables values $v'$, after the occurrence of the event. Additionally, it may contain new local variables (parameters) $vl$. We illustrate the format of an event below:

$$E = \text{ANY } vl \text{ WHERE } G(c, s, vl, v) \text{ THEN } S(c, s, vl, v, v') \text{ END}$$

Event parameters and guards may be sometimes absent, in which case we respectively write ‘WHEN’ instead of ‘ANY vl WHERE’ and ‘BEGIN’ instead of ‘ANY vl WHERE $G(c, s, vl, v)$ THEN’.

The guard $G$ is the necessary condition under which an event might occur; if the guard holds, we call the event enabled. The substitution $S$ is expressed as either a deterministic or a non-deterministic assignment to the variables. $S$ is often referred to as the action part of the event. The action determines the way in which the state variables change when the event occurs. For initialising the system, a sequence of actions is defined. When the guards of several events hold at the same time, then only one event is non-deterministically chosen for execution. If some events have no variables in common and are enabled at the same time, then they can be considered to be executed in parallel since their sequential execution in any order gives the same result.

A model is developed by a number of correctness preserving steps called refinements. One form of model refinement can add new data and new behaviour events on top of the already existing data and behaviour but in such a way that the
introduced behaviour does not contradict or take over the abstract machine behaviour. In 
this superposition refinement (Katz, 1993; Back and Sere, 1996), we present events in 
a refined model by using the shorthand notation “refined_event extends abstract_event”. 
The meaning of this notation is that the refined event is created from the abstract one by 
simply adding new guards and/or new actions. Only the added elements are shown in the 
extended event, while the old guards and actions are implicitly present. In addition to the 
superposition refinement we may also use other refinement forms, such as algorithmic 
refinement (Back and Sere, 1989). In this case, an event of an abstract machine can be 
refined by several corresponding events in a refined machine. This will model different 
branches of execution, that can, for instance, take place in parallel and thus improve the 
algorithmic efficiency.

2.2 The modularisation extension

Recently, the Event-B language and tool support have been extended with a 
possibility to define modules (Iliasov et al., 2010, 2011; Rodin Modularisation Plug-in, 
2011) – i.e., components containing groups of callable atomic operations. Modules 
can have their own (external and internal) state and invariant properties. An important 
characteristic of modules is that they can be developed (refined) separately and, when 
needed, composed with the main system.

A module description consists of two parts – a module interface and a module body. 
Let \( M \) be a module. A module interface \( MI \) is a separate Event-B component. It allows 
the user of the module \( M \) to invoke its operations and observe the external variables 
without having to inspect the module implementation details. \( MI \) consists of external 
module variables \( w \), constants \( c \), sets \( s \), the external module invariant \( M_{\text{Inv}}(c, s, w) \), 
and a collection of module operations; the latter can have their own local variables \( p \) 
and are characterised by their pre- and postconditions, as shown in Figure 1.

**Figure 1** Interface component

<table>
<thead>
<tr>
<th>Interface ( MI )</th>
<th>Sees ( MI_{\text{Context}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables ( w )</td>
<td>Invariants ( M_{\text{Inv}}(c, s, w) )</td>
</tr>
<tr>
<td>Initialisation . .</td>
<td>Process</td>
</tr>
</tbody>
</table>
| \( \text{\begin{tabular}{l}
\text{\( PE_i = \text{ANY } vl \) WHERE } g(c, s, vl, w) \text{ THEN } S(c, s, vl, w, w') \text{ END} \\
\end{tabular}} \) | \( \ldots \) |
| Operations | \( \text{\begin{tabular}{l}
\text{\( O_i = \text{ANY } p \) \text{ PRE } PRE(c, s, vl, w, p) \text{ POST } POST(c, s, vl, w, w', p) \text{ END} } \\
\end{tabular}} \) | \( \ldots \) |

In addition, a module interface description may contain a group of standard Event-B 
events under the Process clause. These events model the autonomous thread of control 
of the module, expressed in terms of their effect on the external module variables. In 
other words, the module process describes how the module’s external variables may 
change between operation calls.
A formal module development starts with the design of an interface. Once an interface is defined, it cannot be altered afterwards. This ensures that a module body may be constructed independently from a model relying on the module interface. A module body is an Event-B machine. It implements the interface by providing a concrete behaviour for each of the interface operations. A set of additional proof obligations are generated to guarantee that each interface operation has a suitable implementation.

When the module $M$ is imported into another Event-B machine (specified by a special clause \textit{USES}), the importing machine can invoke the operations of $M$ and read the external variables of $M$. To make a module specification generic, in $MI_{Context}$ we can define some constants and sets (types) as parameters. The properties over these sets and constants define the constraints to be verified when the module is instantiated. The concrete values or constraints needed for module instantiation are supplied in the \textit{USES} clause of the importing machine.

Module instantiation allows us to create several instances of the same module which are distinctive namely using the clause \textit{prefix}. Different instances of a module operate on disjoint state spaces. Via different instantiations of generic parameters the designers can easily accommodate the required variations when developing components with similar functionality. Hence module instantiation provides us with a powerful mechanism for reuse.

The latest developments of the modularisation extension also allows the developer to import a module with a given concrete set as its parameter. This parameter becomes the index set of module instances. In other words, for each value from the given set, the corresponding module instance is created. Since each module instance operates on a disjoint state space, parallel calls to operations of distinct instances are possible in the same event.

A general strategy of a distributed system development in Event-B is to start from an abstract centralised specification and incrementally augment it with design-specific details. When a suitable level of details is achieved, certain events of the specification are replaced by the calls of interface operations and variables are distributed across modules. As a result, a monolithic specification is decomposed into separate modules. Since decomposition is a special kind of refinement, such a model transformation is also correctness-preserving. Therefore, refinement allows us to efficiently cope with the complexity of distributed systems verification and gradually derive an implementation with the desired properties and behaviour.

2.3 The recovery algorithm in WSANs

WSANs are a rather new generation of sensor networks (Akyildiz and Kasimoglu, 2004), made of two kinds of nodes: sensors and actors. In a WSAN, sensors detect the events that occur in the field, gather them and transmit the collected data to actors. The actors react to the events in the environment based on the received information. The sensor nodes are low-cost, low-power devices equipped with limited communication capabilities, while the actor nodes are usually mobile, more sophisticated and powerful devices compared to the sensor nodes. In addition, the density of sensor nodes in WSANs is much bigger than that of actor nodes. WSANs are dynamic networks where the network topology continuously changes because some new links or nodes are added, or are removed due to hardware crashes, lack of energy, malfunctions, etc.
WSANs have been applied in a variety of commercial, industrial and military applications to react to the situations sensed in the environment. For example, WSANs are installed in forests to prevent and/or resolve forest fires; the firefighter-actors are expected to stop the spread of the fire immediately. Therefore, in real-time applications of WSANs, a fast and effective response is a key concern that can only be provided with a reliable actor-actor coordination. The actor-actor coordination mechanisms provide means for actors to share information and take decisions on the proper reactions. In order to have real-time actor-actor coordination in WSANs, a self-reconfigurable, reliable and real-time communication approach is necessary. Different real-time communication protocols and a number of self-reconfigurable recovery mechanisms have been proposed in the literature (Imran et al., 2012; Gungor et al., 2008; Ngai et al., 2006). However, in most of the cases, the reliability of the developed recovery mechanisms is justified based on simulation results, that cannot guarantee the correctness of these mechanisms.

In this paper we describe a model of the basic functioning of a WSAN, focusing on the actors and their communication links with each others. Besides modelling basic network functioning, we model an abstract recovery algorithm for failed communication links, generated when an actor fails. The neighbours of a failed actor are able to reestablish communication links among themselves based only on localised knowledge of 1-hop (direct) neighbours and 2-hop neighbours (neighbours of neighbours). In the simulated algorithm (Kamali et al., 2008) as well as in the detailed formal model of the algorithm (Kamali et al., 2010), the reestablished links among the neighbours of a failed node are based on the underlying sensor network. However, in this paper we are focusing on the distributed nature of the recovery and do not go into the full details of the algorithm.

The formalisation of the algorithm for self-recovering actor coordination is proposed in Kamali et al. (2012) using Event-B and the associated Rodin platform. The formalism is used to verify essential properties such as the functional correctness and the termination of the recovery mechanism. An important aspect of the algorithm is that indirect links between actors are built in a distributed manner, thus ensuring the self-recovering of the network. To model the functional correctness of the recovery algorithm, authors use the mathematical concepts of tree and forest. In graph theory, a tree is a graph whose any two vertices are connected by a non-cyclic path, while a forest is a set of disjoint trees. A special data structure is introduced in Kamali et al. (2012) to model a forest and prove the correctness properties.

3 Formal specification of the network recovery algorithm

In this section, we present a model of the recovery algorithm in two abstraction steps.

3.1 The initial model

In the context of our initial model we define the finite and non-empty set \( \text{NODE} \) of all the network nodes. These nodes can be either sensor nodes or actor nodes, hence we partition the set \( \text{NODE} \) into the subsets \( \text{sensors} \) and \( \text{actors} \). We also define the generic set \( \text{STATUS} = \{ \text{ok}, \text{fail} \} \), where the constant \( \text{fail} \) denotes the failed status of
a node and the constant \( \text{ok} \) denotes the non-failed status of a node. The initial status of the nodes is defined with the constant function \( \text{initial\_status} \in \text{NODE} \rightarrow \text{STATUS} \).

In the machine of our initial model we define four variables. The status of each node (non-failed or failed) is modelled with the function \( \text{Status} \in \text{NODE} \rightarrow \text{STATUS} \). The relation \( \text{ANET} \in \text{actors} \leftrightarrow \text{actors} \) denotes the bidirectional, non-failed actor links. This relation is irreflexive and symmetric. The \( \text{ANET} \) relation stores only the direct links between actors. For our development purposes, we define the variable \( \text{recovered\_ANET} \in \text{actors} \leftrightarrow \text{actors} \) to store the direct links in \( \text{ANET} \) together with the indirect links that are established by the recovery mechanism. The relation \( \text{recovered\_ANET} \) is irreflexive and symmetric as well. The set \( \text{FailedNodeNeigh} \subseteq \text{actors} \) models non-failed actors, more precisely the actor neighbours of a failed actor. This set is updated when an actor is detected as failed, as shown shortly. We use the annotation \( @\text{acti} \) and \( @\text{grdj} \) to be able to refer to the different actions \( i \) and guard conditions \( j \) of an event.

The \text{INITIALISATION} event sets the status of all the nodes based on the \text{initial\_status} constant function. The \text{ANET} and \text{recovered\_ANET} relations are set to be empty. The set \( \text{FailedNodeNeigh} \) is set to \( \emptyset \).

\begin{verbatim}
INITIALISATION
BEGIN
@act1 Status := initial\_status
@act2 ANET := ∅
@act3 recovered\_ANET := ∅
@act4 FailedNodeNeigh := ∅
END
\end{verbatim}

Except the initialisation, the events in the initial model activate actor nodes from failed to non-failed (\text{ActivateNode}), add actor links (\text{AddLink}), deactivate actor nodes and remove their corresponding links (\text{DeactivateNode}), and abstractly recover connections when an actor fails (\text{Recovery1} and \text{Recovery2}). Thus, starting from an arbitrary, non-deterministic set of failed and non-failed actors that have no links with each other, as established by the \text{INITIALISATION} event, we can randomly activate actors, add links between non-failed actors, and deactivate actors and remove their corresponding links. The latter event models actor failures and enables our recovery mechanism. Until the recovery is complete, the first three events (\text{ActivateNode}, \text{AddLink}, and \text{DeactivateNode}) are not enabled anymore. We thus have a normal operation phase of the algorithm, when the \text{ActivateNode}, \text{AddLink}, and \text{DeactivateNode} events are non-deterministically executed and a recovery phase of the algorithm, when only \text{Recovery1} and \text{Recovery2} events are executed. The phase separation is modelled using the \text{FailedNodeNeigh} variable. While \( \text{FailedNodeNeigh} = \emptyset \), the algorithm is in its operational phase. While \( \text{FailedNodeNeigh} \neq \emptyset \), the algorithm is in its recovery phase. A separate event \text{Recovery3} is defined to complete the recovery phase.

In the \text{ActivateNode} event, the status of a failed \( (@\text{grd2}) \) actor \( (@\text{grd1}) \) is changed to non-failed \( (@\text{act1}) \). The event is enabled if \( \text{FailedNodeNeigh} = \emptyset \) \( (@\text{grd3}) \), as this is an operational event.
A distributed design of a network recovery algorithm

ActivateNode
ANY n WHERE
@grd1 n ∈ actors
@grd2 Status(n) = fail
@grd3 FailedNodeNeigh = ∅
THEN
@act1 Status(n) := ok
END

In the AddLink event we add a link between two distinct, non-failed actors (@grd1-3) that are not connected (@grd4) and are within communication range of each other (@grd5). Here, $r_a$ is the communication range among two actors and $\text{dist}(u \rightarrow v)$ denotes the Euclidean distance between the actors $u$ and $v$. Namely, if $u_x$ and $u_y$ are the horizontal and vertical Cartesian coordinates of the node $u$, respectively, and $v_x$ and $v_y$ are the horizontal and vertical Cartesian coordinates of the node $v$, respectively, then $\text{dist}(u, v) = \sqrt{(u_x - v_x)^2 + (u_y - v_y)^2}$. As ANET is symmetric, we add the link in both directions in $\text{ANET} (@act1)$ and remove it (in both directions) from $\text{recovered}_{\text{ANET}}$. This corresponds to cancelling the (potential) temporary links proposed by the recovery algorithm. When two nodes can be connected directly, an indirect link consuming more power is not needed. Therefore, the constructed indirect links are removed from $\text{recovered}_{\text{ANET}}$. The event is enabled only if $\text{FailedNodeNeigh} = \emptyset (@grd6)$, as this is an operational event.

AddLink
ANY n m WHERE
@grd1 n ∈ actors ∧ m ∈ actors
@grd2 Status(n) = ok ∧ Status(m) = ok
@grd3 n ≠ m
@grd4 n \rightarrow m \notin \text{ANET}
@grd5 dist(n \rightarrow m) < r_a
@grd6 FailedNodeNeigh = ∅
THEN
@act1 \text{ANET} := \text{ANET} \cup \{n \rightarrow m, m \rightarrow n\}
@act2 \text{recovered}_{\text{ANET}} := \text{recovered}_{\text{ANET}} \setminus \{n \rightarrow m, m \rightarrow n\}
END

The DeactivateNode event changes the status of a non-failed actor (@grd1-2) to that of a failed one (@act1); also, all the links of that actor are removed from both $\text{ANET}$ and $\text{recovered}_{\text{ANET}}$, expressed with the domain subtraction operator $\setminus$ and the range subtraction operator $\rightarrow$ (@act2-3). The domain and range subtraction operators on $\text{ANET}$ and $\text{recovered}_{\text{ANET}}$ relations denote only those pairs whose first and second element is not $n$. In addition, the neighbours of the failed actor become members of the $\text{FailedNodeNeigh}$ set (@act4).

The DeactivateNode event is an operational one, enabled only when $\text{FailedNodeNeigh} = \emptyset (@grd3)$. If the failing actor had neighbours in $\text{ANET}$, then $\text{FailedNodeNeigh} \neq \emptyset$ after @act4 is executed. At this point, the algorithm has entered into the recovery phase. In our algorithm, we model the situation where more than one actor can fail at a time only if these actors have no neighbours in $\text{ANET}$. If a node with neighbours fails, then no other node can fail again until we have recovered the communication between the neighbours of the failed node. This is modelled by
the DeactivateNode event not being enabled again until FailedNodeNeigh becomes empty again (@grd3).

```
DeactivateNode
ANY n WHERE
@grd1 n ∈ actors
@grd2 Status(n) = ok
@grd3 FailedNodeNeigh = ∅
THEN
@act1 Status(n) := fail
@act2 ANET := {n} ↦ ANET \ {n}
@act3 recovered_ANET := {n} ↦ recovered_ANET \ {n}
@act4 FailedNodeNeigh := ANET \ {n}
END
```

When an actor with neighbours has been deactivated, we need to check whether a recovery is needed. The event Recovery1 is enabled when two neighbours of a deactivated actor (@grd1) have no short connection through other neighbours (@grd2) (i.e., there is no path from one actor to the other at most 2-hop long). We notice that this check does not imply that the deactivation of an actor has led to partitioning the actor network, although in some cases, it may have led. If Recovery1 is enabled, then a direct actor-actor link is established and stored separately from ANET, in recovered_ANET (@act1). This separation is essential because ANET models only the real direct links among nodes. The recovered links modelled by recovered_ANET are not direct links, i.e., nodes are not within communication range of each other; they can typically only communicate through some intermediate nodes which are not modelled at this abstract level. As FailedNodeNeigh is a subset of the finite set NODE, we observe that the Recovery1 event can be enabled only a finite number of times, hence the recovery phase terminates. Technically, this is true because card(FailedNodeNeigh) decreases at each execution of Recovery1 and eventually the guard of Recovery1 will hold no longer. We observe that node n cannot be equal to node k due to @grd2 (ANET; ANET is reflexive). ANET; ANET denotes the forward relational composition of ANET by itself.

```
Recovery1
ANY n k WHERE
@grd1 n ∈ FailedNodeNeigh ∧ k ∈ FailedNodeNeigh ∧ n ≠ k
@grd2 n ↦ k, k ↦ n \∈ ANET ∧ n ↦ k \∈ ANET; ANET
THEN
@act1 recovered_ANET := recovered_ANET \ {n ↦ k, k ↦ n}
@act2 FailedNodeNeigh := FailedNodeNeigh \ {n}
END
```

```
Recovery2
ANY n k WHERE
@grd1 n ∈ FailedNodeNeigh ∧ k ∈ FailedNodeNeigh
@grd2 n ↦ k ∈ ANET ∨ n ↦ k ∈ (ANET; ANET) \ (actors ⇉ id)
THEN
@act1 FailedNodeNeigh := FailedNodeNeigh \ {n}
END
```
The Recovery2 event treats the situation when a failure is detected but an alternative path through 1-hop or 2-hop neighbours already exists between the neighbours of the deactivated actor \((n\rightarrow k \in ANET \vee n \rightarrow k \in ANET; ANET)\). In this case, FailedNodeNeigh is simply updated. In Recovery2 we are also sure that \(n\) and \(k\) cannot be equal (@grd2). The last element of FailedNodeNeigh is removed via the Recovery3 event.

Overall, in this model we have described the non-deterministic activation and deactivation of actor nodes and the adding and removing of actor links in a dynamic (wireless sensor-actor) network for whom a communication problem among the actors can be detected and recovered from via direct actor links. The recovery algorithm assumes some global network knowledge for the recovery, expressed by \(ANET; ANET\). Also, the recovery mechanism establishes direct links among the non-failed actor neighbours of the failed actor. We observe that both the recovery assumption and the recovery mechanism can be used in practice only for strategic actors, i.e., actors whose range is sufficiently large to check the contents of \(ANET; ANET\) and establish direct actor links. The following model considers more localised assumptions as well as indirect recovery paths.

3.2 The refined model

In the initial model we have defined the actor network recovery based on global knowledge about the whole network \((ANET; ANET)\). In this model we assume that each actor has access only to information about its 1-hop neighbours and 2-hop neighbours, i.e, we restrict the actor-actor communication recovery to take place via local information. This model is a refinement of the initial model. The goal of this refinement step is to supplement the global knowledge of the network in the initial model \((ANET; ANET)\) with localised knowledge for every actor. We achieve this by introducing a new variable, the irreflexive relation \(l_{net} \in (\text{actors} \times \text{actors}) \leftrightarrow \text{NODE}\) that, for each actor, keeps track of the 1-hop and 2-hop neighbours and the recovered links modelled by recovered\(_{ANET}\) in the initial model. The relation \(L_{net}\) describes all these localised links between nodes. Those 2-hop links that are reestablished via nodes other than actors denote the recovered\(_{ANET}\) links of the initial model.

When a new link is added between two actors, the \(L_{net}\) relation also needs to be updated. Therefore, the AddLink event is extended to also add links to \(l_{net}\). For every two actors \(n\) and \(m\) that have a direct link, \(n \rightarrow m \rightarrow m\) and \(m \rightarrow m \rightarrow n\) are added in
$l_{net}$ (@act3), meaning that $n$ has a link with $m$ through $m$ ($m$ is a 1-hop neighbour of $n$) and $m$ has a link with $n$ through $n$ ($n$ is a 1-hop neighbour of $m$). Moreover, the indirect links between $n$ and $m$ through sensors (denoted by expression such as $\{m\} \times \{n\} \times (\text{Status}^{-1}\{\text{ok}\}) \setminus \text{actors}$) are removed from $l_{net}$ because there is now a direct link with less power consumption. This is similar to @act2 of this event in the initial model.

$\text{AddLink}$

extends AddLink

THEN

@act3 $l_{net} := l_{net} \cup \{n \rightarrow m \rightarrow m \rightarrow n \rightarrow n\} \setminus ((\{n\} \times \{m\} \times \text{Status}^{-1}\{\text{ok}\}) \setminus \text{actors} ) \cup (\{m\} \times \{n\} \times \text{Status}^{-1}\{\text{ok}\} \setminus \text{actors})$

END

$\text{Add}_n_{net2hopLink}$

ANY n m k WHERE

@grd1 Status(n) = ok \wedge Status(m) = ok \wedge Status(k) = ok

@grd2 n \rightarrow k \rightarrow k \in l_{net} \n \rightarrow m \rightarrow m \in l_{net} \wedge

n \rightarrow k \rightarrow m \notin l_{net} \wedge k \rightarrow n \rightarrow m \notin l_{net}

@grd3 m \neq n \wedge n \neq k

@grd4 FailedNode:Neigh = \emptyset

THEN

@act1 $l_{net} := l_{net} \setminus ((\{n\} \times \{k\} \times \text{Status}^{-1}\{\text{ok}\}) \setminus \text{actors} ) \cup (\{k\} \times \{n\} \times \text{Status}^{-1}\{\text{ok}\} \setminus \text{actors})$

END

The $\text{Add}_n_{net2hopLink}$ event is a newly introduced event that handles the addition of 2-hop neighbour links for actors. If an actor has a direct link with two other actors, then these actors will be 2-hop neighbours of each other. The actors involved in this event have to be non-failed (@grd1), have a direct (1-hop) link but not yet a 2-hop link through actors in $l_{net}$ (@grd2) and be distinct (@grd3). Also, this new event belongs to the operational phase of the algorithm, hence, $\text{FailedNode:Neigh} = \emptyset$ is part of its guard (@grd4). If these conditions are satisfied, then any 2-hop links between $n$ and $k$ reestablished through sensors are removed from $l_{net}$ and the detected 2-hop links through actors are added in $l_{net}$ in both directions (@act1).

When deactivating an actor node, all its links are also removed. Thus, in the $\text{DeactivateNode}$ event the new action @act4 removes all the links to and from the failed actor in the $l_{net}$ relation. The expression $\{n\} \times \text{dom}(ANET) \times \text{Status}^{-1}\{\text{ok}\}$ describes all the links from $n$, either direct connections (1-hop neighbours) or indirect connections (2-hop neighbours) and the expression $\text{dom}(ANET) \times \{n\} \times \{n\}$ describes all the direct connections (1-hop neighbours) links to $n$.

$\text{DeactivateNode}$

extends DeactivateNode

THEN

@act4 $l_{net} := l_{net} \setminus ((\{n\} \times \text{dom}(ANET) \times \text{Status}^{-1}\{\text{ok}\}) \cup (\text{dom}(ANET) \times \{n\} \times \{n\})$

END
We now need to model the detection of failed actors and the recovery of links based on local information instead of being based on the global actor-actor coordination as described by ANET; ANET. We now use $l_{net}$ information in addition to ANET for detecting an actor failure (@grd3) and recovering links in the Recovery1 event.

\[
\text{Recovery1}
\]

\[
\text{Recovery1} \text{ extends Recovery1}
\]

\[
\text{ANY } n \text{ m k WHERE }
\]

\[
\text{@grd3 } n \rightarrow k \rightarrow m \in l_{net} \land k \rightarrow n \rightarrow m \in l_{net} \land n \rightarrow m \rightarrow m \notin l_{net} \land
\]

\[
k \rightarrow m \rightarrow m \notin l_{net} \land n \rightarrow k \rightarrow k \notin l_{net} \land k \rightarrow n \rightarrow n \notin l_{net}
\]

\[
\text{THEN }
\]

\[
\text{@act3 } l_{net} \cup l_{net'} \subseteq (l_{net} \setminus \{n \rightarrow k \rightarrow m, k \rightarrow n \rightarrow m\}) \cup
\]

\[
\{(k \times \{n\}) \times Status^{-1}([ok])\} \cup \{(n \times \{k\}) \times Status^{-1}([ok])\}
\]

\[
\cup \text{indir}(k, n) \cup \text{indir}(n, k) \cup \text{indir}(n, k) \cup \text{indir}(n, k)
\]

\[
\text{END}
\]

\[
\text{Recovery2}
\]

\[
\text{Recovery2} \text{ extends Recovery2}
\]

\[
\text{ANY } n \text{ m k WHERE }
\]

\[
\text{@grd3 } (n \rightarrow k \rightarrow m \in l_{net} \land n \rightarrow m \rightarrow m \notin l_{net} \land k \rightarrow n \rightarrow m \in l_{net}
\]

\[
\land k \rightarrow m \rightarrow m \notin l_{net}) \lor (n \rightarrow k \rightarrow k \in l_{net} \land k \rightarrow n \rightarrow n \in l_{net})
\]

\[
\text{THEN }
\]

\[
\text{@act2 } l_{net} := l_{net} \setminus \{n \rightarrow k \rightarrow m, k \rightarrow n \rightarrow m\} \cup \text{indir}(n, m) \cup \text{indir}(k, m)
\]

\[
\text{END}
\]

When actor $m$ is detected as failed, neighbours of $m$ ($n$ and $k$) that have a connection with each other through $m$ ($n \rightarrow k \rightarrow m$, $k \rightarrow n \rightarrow m \in l_{net}$) need to find an alternative path towards each other. If there is no other route in $n \rightarrow m \rightarrow m, k \rightarrow n \rightarrow m, n \rightarrow k \rightarrow k$ and $k \rightarrow n \rightarrow n \notin l_{net}$, then $l_{net}$ should be updated by removing invalid links and adding new routes.

Since $m$ is failed, links between $n$ and $k$ through $m$ are not anymore valid, so $n \rightarrow k \rightarrow m$ and $k \rightarrow n \rightarrow m$ are removed from $l_{net}$. Then we need to add new links to connect $n$ and $k$. In this refinement, we model that actor $n$ can establish a link with actor $k$ through any non-failed node: $\{n\} \times \{k\} \times Status^{-1}([ok])$ and similarly for actor $k$ to establish a new link with actor $n$: $\{k\} \times \{n\} \times Status^{-1}([ok])$. For clarity, we use the following short definitions: $\text{indir}(k, n) = ANET[k] \times n \times k$ and $\text{indir}(n, k) = n \times ANET[k] \times k$. When node $n$ establishes a link with $k$, neighbours of $n$ also need to add node $k$ to their 2-hop neighbours list ($\text{indir}(n, k)$ and $\text{indir}(n, k)$). Moreover, neighbours of $k$ need to add $n$ to their 2-hop neighbours list ($\text{indir}(k, n)$ and $\text{indir}(n, k)$). The updating process of $l_{net}$ is described by @act3 in the Recovery1 event.

We also add a new action to the event Recovery2 that updates $l_{net}$ by removing all the links to the failed actor or through it.

The relation $l_{net}$ is an elegant and very abstract data structure relating two actor nodes in its domain via a third node in its range. Indirect links between actors are now established non-deterministically based on localised information. The graphical overview of our model refinement is shown in Figure 2, where ‘Initial Model’ denotes the model described in Section 3.1 and ‘refined model’ denotes the model described in the current Section 3.2.
4 Distributing the recovery algorithm

In this section we put forward the decomposition of our centralised WSAN model into a model for a WSAN infrastructure and a model for the distributed actor node(s). The correctness of this decomposition step is addressed as a special kind of model refinement.

4.1 A distributed model of the actor network

In the initial model we capture the recovery algorithm in a centralised way. This centralisation is achieved by allowing the actor nodes to access global network links, modelled by $\text{ANET}$. However, in the following refinement step, the variable $l_{\text{net}}$ is introduced to limit the actor access to the local network links rather than the global ones. Such a refinement step allows us to express the distributed nature of the recovery algorithm in WSANs. Specifically, the refined model represents the communicating infrastructure together with actions of separate actors, so that everything is modelled in a single machine. On one hand, the modelling and the verification of properties in the network is simpler but, on the other hand, it causes difficulties for its distributed implementation. Therefore, in order to implement a distributed actor network, we need to separate the WSAN infrastructure and the individual actor operations.

The overall scheme of the decomposition refinement is shown in Figure 3. The model presented in Section 3.2 (‘refined model’ in Figure 3) is now split into two parts: an interface model, named ‘Actor Interface’ and a machine model, named ‘WSAN Middleware’. The interface specifies the external state and the behaviour model of actor nodes, while the machine models the WSAN infrastructure that enables the nodes to communicate with each other. The actor interface is imported into the machine as a module, which is indexed by the set of actor nodes. This machine acts as the communication middleware providing access to module instances via the given interface.
4.2 Actor interface

The actor interface defines several external variables that can be accessed for reading by the middleware. These variables model the visible state of an actor node. Each actor interface has four variables: `actor_state`, `one_hop`, `two_hop` and `n_n`. The `actor_state` variable represents the state of a node, which can be either `ok` or `fail`. This variable refines the corresponding variable `Status` of the abstract model. The `one_hop`
and two\text{hop} variables store the direct and indirect neighbours of each node, thus corresponding to ANET and l\text{net}. The \text{nn} variable is a control variable that informs the WSAN middleware about recent state changes in an actor. If the value of \text{nn} is \text{TRUE}, this means that the node has recently became active and thus the link discovery should be started. Otherwise, the node is either inactive or the link discovery process has already been done.

\begin{align*}
\text{actor\_state} & \in \text{STATUS} \\
\text{one\_hop} & \subseteq \text{actors} \\
\text{two\_hop} & \in \text{actors} \leftrightarrow \text{actors} \\
\text{nn} & \in \text{BOOL}
\end{align*}

\begin{align*}
\text{PROCESS} \\
\text{actor\_active} \\
\text{WHEN} & \hspace{1em} \text{actor\_state} = \text{fail} \\
\text{THEN} & \hspace{1em} \text{actor\_state} := \text{ok} \\
& \hspace{1em} \text{nn} := \text{TRUE} \\
\text{END}
\end{align*}

\begin{align*}
\text{PROCESS} \\
\text{actor\_inactive} \\
\text{WHEN} & \hspace{1em} \text{actor\_state} = \text{ok} \\
\text{THEN} & \hspace{1em} \text{actor\_state} := \text{fail} \\
& \hspace{1em} \text{nn} := \text{FALSE} \\
\text{END}
\end{align*}

\begin{align*}
\text{OPERATION} \\
\text{Add\_Actor\_Link} \\
\text{ANY} \\
\text{new\_node} \\
\text{PRE} \\
& \hspace{1em} \text{actor\_state} = \text{ok} \\
& \hspace{1em} \text{new\_node} \notin \text{one\_hop} \\
\text{RETURN} \\
& \hspace{1em} \text{r} \\
\text{POST} \\
& \hspace{1em} \text{one\_hop}' = \text{one\_hop} \cup \{\text{new\_node}\} \\
& \hspace{1em} \text{two\_hop}' = \text{two\_hop} \setminus \{\text{new\_node}\} \\
& \hspace{1em} \times (\text{NODE} \setminus \text{actors}) \\
& \hspace{1em} \text{r}' = \text{one\_hop} \\
\text{END}
\end{align*}

Each actor can have its own autonomous process which is not dependent on the rest of the network. The events describing such a process are given in the PROCESS part of the actor interface. In our model, we have two events, \text{actor\_active} and \text{actor\_deactive}, that change the state of a node from inactive to active and from active to inactive, respectively. In addition, the control variable is set to inform the communication middleware about the recent change in the node state.

In the operation part of the actor interface, the operations that can be called by the middleware are defined. The middleware calls these operations in its events to add direct and indirect links between nodes as well as recover from disconnectivity. In our interface, we have six operations where can be called by the communication middleware. However, only the three operations referring to the distributed aspect of the network recovery algorithm are discussed in this paper, for simplicity. We mention some of the other operations briefly when called in the events of the middleware. The
Add\_Actor\_Link adds a new link to a given node into the variable one\_hop. The node is passed as a parameter of the operation call. Moreover, the call removes all the indirect links via sensors for the given node from the variable two\_hop. This is because a direct actor link removes the need for indirect ones.

The Add2hop\_Link operation updates the two\_hop variable by adding a new link, passed as the operation parameter and returns the void value to the middleware.

\begin{verbatim}
OPERATION
  Recovery
  ANY
  m fn
  PRE
    actor\_state = ok \land m \in one\_hop
  POST
    one\_hop' = one\_hop \setminus \{m\}
    (fn \notin one\_hop \land fn \notin \text{dom}(two\_hop \cup \{m\}) \Rightarrow
      two\_hop' \subseteq (two\_hop \cup \{m\}) \cup \{fn\} \times (\text{NODE} \setminus \text{actor})
    (fn \in one\_hop \land fn \in \text{dom}(two\_hop \cup \{m\}) \Rightarrow two\_hop' = two\_hop \cup \{m\})
\end{verbatim}

The Recovery operation removes invalid links and adds a new indirect link, if needed. The failed node \( m \) which is passed as an argument to the Recovery operation is removed from the one\_hop list. Moreover, a neighbour \( fn \) (a node in the FailedNodeNeigh set in the previous models) of the failed node \( m \) is passed to the Recovery operation in order to reestablish a link through sensors, between the actor and \( fn \). In the Recovery operation we then check the one\_hop and two\_hop lists and if there is a link between the actor and \( fn \), two\_hop list is just updated by removing all indirect links through the failed node. If there is a link neither in one\_hop list nor two\_hop list, two\_hop list is updated by removing all indirect links through the failed node \( m \) and adding an indirect link to \( fn \) through sensor nodes.

4.3 WSAN middleware

In this section, we show how the defined interface can be used in the refined machine. Moreover, we discuss how the refined machine can act as a WSAN middleware.

\begin{verbatim}
USES
  actor\_interface with prefix actor

PROCESS LINK
  actor\_active : ActivateNode
  actor\_inactive : DeactivateNode
\end{verbatim}

In the machine part of the refined model, we import the actor interface indexed by the actor set. As explained in Section 2.2, in this way we introduce a number of module instances, each for a particular actor node in the network. In order to refine the abstract events of the model before decomposition, in some cases we need to link these events to the events that are now distributed among the module instances. Specifically,
ActivateNode and DeactivateNode are the abstract events that are linked to the corresponding events in the process part of the actor interface.

One of the goals of the decomposition refinement step is data refinement of the data structures modelling the actor coordination and recovery mechanism in WSANs by the ones representing the coordination model of individual actors. Therefore, we need to show that the one_hop and two_hop variables of the actor module instances are correct replacements (i.e., data refinements) of the \( l_{\text{net}} \) variable from the abstract model. In addition, the abstract Status variable is now data refined by actor_state residing in module instances. The gluing invariants between abstract and concrete variables are as follows:

\[
\begin{align*}
l_{\text{net}} &= (\text{id} < (\text{one} \text{hop} \times \text{one} \text{hop})) \cup \text{two} \text{hop} \\
\text{status} &= \text{actor} \text{state}
\end{align*}
\]

The remaining events of the refined machine can be split into two groups. The first group consists of a single (new) event describing how the middleware monitors the actor nodes, specifically looking for new activated nodes or previously active and currently failed nodes. The second group consists of several events specifying the middleware reactions in response to the detected changes. The corresponding reactions are specified using operation calls to the affected node instances. These events are refinements of the abstract events for adding new links and initiating the recovery mechanism.

**EVENTS**

\[
\begin{align*}
\text{ReadHeartBeat} & \quad \text{ANY} \\
n & \quad \text{new fail} \\
\text{WHERE} & \\
n & \in \text{actors} \\
(n,n(n) = \text{TRUE} \land \text{new_node} = \emptyset \land \text{new} = n) \lor (\text{new} = \emptyset) \\
(\text{actor_state}(n) = \text{fail} \land \text{one_hop}(n) \neq \emptyset \land \text{inactive_node} = \emptyset \land \text{fail} = n) \\
\lor (\text{fail} = \emptyset) \\
\text{THEN} & \\
\text{new_node} & := \text{new} \\
\text{inactive_node} & := \text{fail} \\
\text{FailedNodeNeigh} & := \text{DeactivateUpdate}(\text{fail}) \\
\end{align*}
\]

The new monitoring event is called ReadHeartBeat. As node activation and deactivation independently happens in a node and the network is not immediately aware of these changes, this event alarmsg the communication middleware for any changes in the network topology. The detected changes are stored into two new variables new_node and inactive_node, which represent the indexes of, respectively, a node that was recently activated in the network and a node that recently failed (i.e., became inactive). In its guard, the ReadHeartBeat event checks the state of actor node instances. If the \( n,n \) value of the actor node is \( \text{TRUE} \) then new_node is updated to the actor index. Moreover, if the actor_state value of the actor is \( \text{fail} \) and one_hop is not empty, i.e., the actor node has recently failed and recovery should be started, then the inactive_node variable is also updated by the actor index. Finally, the DeactivateUpdate operation of the failed node is called to set the list of one_hop and two_hop with \( \emptyset \) and return the neighbours of the node to update FailedNodeNeigh. When one_hop list
of a failed node is set to $\emptyset$ denotes that the node failure has been considered and the recovery links had been reestablished after detection of the failed node. Guards of the \texttt{ReadHeartBeat} event corresponds to conditions that if a node has recently added or removed from the network which are modelled by evaluating the value of $n.n$ and \texttt{one-hop} in each node, respectively.

The second group of events consists of the events \texttt{AddLink}, \texttt{Add2hopLink}, and \texttt{Recovery}. The \texttt{AddLink} event is enabled when a new node joins the network and an active actor is in its communication range. By calling the \texttt{Actor_AddLink} operations of these two neighbours, the corresponding internal lists of 1-hop neighbours of these nodes are updated, while the lists of 2-hop neighbours are returned to the \texttt{AddLink} event. The returned lists are saved into two variables $m.n_{neigh}$ and $n.n_{neigh}$ to be used in the \texttt{Add2hopLink} event to update the lists of 2-hop neighbours. Note the syntax of operation calls including two pairs of parentheses: in the first parentheses, the index of a called node instance is given, while, within the second parentheses, the concrete arguments of the procedure call are passed to the given module instance.

In the \texttt{Add2hopLink} event, the \texttt{Add2hopLink} operation is called for different module instances and their results are returned to the event. Since the return value of the \texttt{Add2hopLink} operation is \texttt{void}, we have to assign it to the corresponding variables of the type \texttt{VOID}. However, for the sake of clarity, we introduce a shorthand notation for such calls: instead of \texttt{void var := Operation(index) (parameters)} we simply write \texttt{Operation(index) (parameters)}. Please also note that the module instance index values used within operation calls in this event are actually not single values but sets of indexes. In such a way, we can specify a multiple (broadcasting) call, when the same operations of a number of affected module instances are called simultaneously. Besides, the $n.n_{Modify}$ operation updates the value of $n.n$ from \texttt{TRUE} to \texttt{FALSE} and is called for \texttt{new_node}. The $n.n$ variable of \texttt{new_node} is updated to denote that the \texttt{one-hop} and \texttt{two-hop} lists of \texttt{new_node} and its neighbours are updated. Updating of $n.n$ variable in a node causes the communication middleware to not consider the node as a new node anymore.

\begin{verbatim}
EVENTS
  AddLink
  ANY
  n m
  WHERE
    n $\in$ new_node
    m $\in$ actors $\land$ n $\neq$ m
    dist(n $\rightarrow$ m) $<$ r$a$
  THEN
    m.n_{neigh} := (Actor_AddLink(m)(n)) $\times$ \{m\}
    n.n_{neigh} := (Actor_AddLink(n)(m)) $\times$ \{n\}
    new_link := \{n, m\}
END

EVENTS
  Add2hopLink
  ANY
  n m
  WHERE
    n $\in$ new_node $\land$ m $\in$ new_link $\land$ n $\neq$ m
  THEN
    Add2hopLink(dom(n.n_{neigh}))({m $\rightarrow$ n})
    Add2hopLink(dom(m.n_{neigh}))({n $\rightarrow$ m})
    Add2hopLink(dom(n))({m.n_{neigh}})
    Add2hopLink(dom(m))({n.n_{neigh}})
    new_link := $\emptyset$
    new_node := $\emptyset$
    n.n_{Modify}(new_node)
END
\end{verbatim}
Finally, the Recovery event is enabled when a node failure is detected by the ReadHeartBeat event. Since the recovery mechanism is now distributed among the actor nodes, in the refined machine we only call the recovery operation of neighbours of a failed node. Therefore, we merge the Recovery1 and Recovery2 events of the abstract machine into the Recovery event of the refined model.

\[
\begin{align*}
\text{EVENTS} & \\
\text{Recovery} & \\
& \text{refines} \\
& \text{Recovery1} \text{ Recovery2} \\
\text{ANY} & \\
& n \ k \\
\text{WHERE} & \\
& n \in \text{FailedNodeNeigh} \\
& k \in \text{FailedNodeNeigh} \\
\text{THEN} & \\
& \text{Recovery}(n)(\text{inactiveNode} \rightarrow k) \\
& \text{Recovery}(k)(\text{inactiveNode} \rightarrow m) \\
& \text{FailedNodeNeigh} := \text{FailedNodeNeigh} \ \backslash \ \{n\} \\
\text{END} & \\
\end{align*}
\]

5 Contribution

In this paper we have employed Event-B to uncover a distributed software design for a network recovery algorithm. This algorithm has been formalised before in Event-B (Kamali et al., 2010) and its correctness and termination properties proved based on the Rodin platform (Kamali et al., 2012). The algorithm deals with recovering communication links between actors when intermediary actors fail. The nature of WSANs imposes that every useful recovery algorithm be distributed, so that it is taken care of by individual actors, based on some given infrastructure. In the previous works on this algorithm (Kamali et al., 2010, 2012), the actor actions and the infrastructure are all integrated in one specification. This obviously simplifies the understanding of the involved mechanisms as well as the formulation and proofs of the properties. However, it is problematic to implement such a specification. We address this difficulty in this paper and propose a distributed design for the network recovery algorithm. As we carry out our derivation still in Event-B, we can keep all the enumerated advantages of simplicity and proving. In addition, we get a distributed design, much easier to implement.

Our approach for distribution follows the object-oriented paradigm. Since there is a number of entities of the same kind in the network, i.e., network nodes, a class (indexed module) of nodes can be developed separately by modelling their local variables and behaviours. Specifically, following our modelling methodology, we introduce a module interface that models the state and the behaviour of individual nodes. In such a way, we map the notion of a class in object-oriented programming to the notion of of an indexed module in Event-B. Similarly to creating new objects of a class in object-oriented programming, in our model we import an interface with an index set as its parameter. Such an approach lets us distribute models of individual nodes in the network so that they can be more transparently transformed into a part of distributed network
implementation. The decomposed model provides enough intuition towards creating object-oriented code out of it.

Another interesting aspect of our derived distributed design refers to the failing actors. Actors can fail and be activated non-deterministically both in the integrated specification and in the distributed one. In the integrated specification however, only one node could fail at a time and the algorithm takes care of recovering its neighbours communication. Only after that can the normal operation of the actor, including other possible actor failures, occur. In the distributed design, any number of actors can fail simultaneously or sequentially. The distributed recovery algorithm takes care of recovering the communication of each failed actor’s neighbours, in a non-deterministic order and (still) one at a time. This is a significant reduction of the previous constraints and constitutes an important contribution to modelling a reliable and more realistic distributed recovery.

6 Related work and conclusions

Several formal developments of complex systems have used the Event-B formal method, for instance network protocols (Abrial et al., 2003; Rehm, 2010), network-on-chip modelling (Kamali et al., 2011a, 2011b) and sensor networks (Kamali et al., 2010). Cansell and Méry (2006) model a development of a distributed reference counting algorithm by using the refinement technique. Hoang et al. (2009) develop a topology discovery algorithm in Event-B, to prove safety and liveness properties. The major contribution in these formal models consists in developing correct-by-construction models with the help of the refinement technique. Our contribution goes one step further, to decompose a verified development towards a distributed implementation.

Ball and Butler (2006) present a practical approach to the formal development of multi-agent systems in Event-B by using abstraction and decomposition techniques. The main purpose of using the decomposition method for the development is to cope with the complexity of the system. When the model is refined, it becomes more concrete and complex. Therefore, decomposition is ideal for simplifying the complexity of the model. In comparison, our aim is to apply the decomposition technique to approach an implementation of a distributed system.

Iliasov et al. (2011) decompose an integrated correct-by-construction specification of a distributed system to gain its distributed program. The paper addresses a state-based formal approach to correct-by-construction development of distributed programs. The authors present their approach by developing of a distributed leader election protocol as a case study. Decomposition refinement in the case study is accomplished by two instances of the module interface while in this paper we use indexed instances and approach an object-oriented implementation.

Ray and Cleaveland (2004) present a formal modelling of middleware-based distributed systems. They extend architectural integration diagrams (AIDs) (Ray and Cleaveland, 2003) by providing an operational semantics that supports formal analysis of distributed system designs. AID provides a mechanism that supports a wide variety of interprocess communication and frees the designers from manually modelling the different interprocess communications. Another middleware-based systems formalism is presented in Dan and Danning (2010). They work on semi-formal UML-based analysis using UML profiles and a notion of behavioural semantics. The main purpose of these
studies is the proposal of an operational semantics for modelling middleware-based
distributed systems to analyse different designs. In this paper, we understand the
middleware as a network infrastructure stepwise derived to achieve distribution.

The derivation of a distributed design from an integrated algorithm is very similar
in nature to well-established research in formal methods, for instance in Back and Sere
(1996). The derivation of a distributed broadcast algorithm from a centralised one is
employed there as a case study and the formal development is carried out in action
systems (Back and Kurki-Suonio, 1983) instead of Event-B. It is worth mentioning that
actions systems are in fact a precursor of Event-B. However, Event-B has the associated
Rodin platform, which makes proving an automated matter and thus brings a significant
advantage to the establishment of reliable practices in software development.

References
development of IEEE 1394 tree identify protocol’, Formal Aspects of Computing,
New York, NY, USA.
in Butler, Michael, Hinchey, Michael, G. and Larrondo-Petrie, Maria, M. (Eds): Formal
University Press, New York, NY.
in Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed
Computing, pp.131–142.
J.L.A. van de Snepscheut (Ed.): Mathematics of Program Construction, 375th Anniversary
of the Groningen University, International Conference, Groningen, The Netherlands, June
on the distributed reference counting algorithm’, Theoretical Computer Science, Vol. 364,
No. 3, pp.318–337.
in Information Science and Engineering (ISISE), 2010 International Symposium on,
pp.213–218.
model decomposition to a space craft system in Event-B’, in M.G. Bobaru, K. Havelund,
G.J. Holzmann and R. Joshi (Eds.): NASA Formal Methods, Vol. 6617 of Lecture Notes in
A distributed design of a network recovery algorithm


