Uppaal vs Event-B for Modelling Optimised Link State Routing

Mojgan Kamali(✉) and Luigia Petre

Åbo Akademi University, Turku, Finland
{mojgan.kamali,lpetre}@abo.fi

Abstract. In this paper we compare models developed in two formal frameworks, Uppaal and Event-B, for the Optimised Link State Routing (OLSR) protocol. OLSR is one of the proactive routing protocols used in Mobile Ad-hoc Networks (MANETs) and Wireless Mesh Networks (WMNs). We also describe different aspects of the Uppaal and Event-B formalisms. This leads to a more general comparison of both formalisms, considering the following criteria: their specification languages, their update of variables mechanism, their modularity methods, their verification strategies, their scalability potentials and their real-time modelling capabilities. Based on it, we provide several guidelines for when to use Uppaal or Event-B for formal modelling and analysis.

1 Introduction

Continuous connectivity is a defining feature of our current working routines as well as of our free-time ones. We expect to be able to access information at all times as well as be able to communicate to various entities at all times. Technically, this is ensured with myriads of interconnected networks that offer us coverage and route all our requests for information and communication in certain ways. Hence, routing is a fundamental stone of our lifestyles and as such, presents enormous interest for study. Routing is obviously not a new concept for the era of continuous connectivity; it has been around since the first networks were developed some decades ago. Along with network evolution, routing has however evolved as well, with numerous algorithms in use today.

Routing protocols are divided into two main categories: proactive and reactive. Proactive protocols select routes in advance, by having network nodes exchanging (control) messages about all the other network nodes. Consequently, an injected data packet can be delivered to the destination immediately. Examples of such protocols are Optimised Link State Routing (OLSR) protocol [10], Better Approach To Mobile Ad hoc Networking (BATMAN) routing protocol [22], etc. Reactive protocols search for routes to destination nodes on demand, whenever a data packet is injected into the network. Examples of reactive protocols are Ad hoc On-Demand Distance Vector (AODV) protocol [23], Dynamic Source Routing (DSR) protocol [14], etc.

In this paper we compare two models for the OLSR proactive protocol. This protocol is used for routing in Wireless Mesh Networks (WMNs). WMNs are

© Springer International Publishing AG 2017
K. Barkaoui et al. (Eds.): VECoS 2017, LNCS 10466, pp. 189–203, 2017.
DOI: 10.1007/978-3-319-66176-6_13

self-healing and self-organising wireless technologies supporting broadband communication without requiring any wired infrastructure. They are employed in a wide range of application areas such as emergency response networks, communication systems, video surveillance, etc. A central feature of a WMN is that its topology, in terms of active nodes and links, can vary quite much. OLSR is adapted to this feature by continuously updating the information that any node has about any other node, based on the most recent 'scanning' of the network. It thus finds good-enough routes to all destinations.

Previously, our goal was to model OLSR and analyse its properties [15,17,18]. There are numerous frameworks and techniques, formal and less formal, that one can choose for modelling purposes. Since we are interested in analysis, formal methods with their underlying mathematical foundations are best suited. However, the question is which formal method to choose. In this paper we resume our experiences with two formal methods, the Uppaal model checker and the Event-B theorem prover.

In Uppaal [7], safety and liveness properties are expressed using Computation Tree Logic (CTL). Constants, data structures and procedures are defined in a C-like language and modularity is addressed via components, represented as timed automata, that communicate with each other via channels. Uppaal has a model checking tool[1], that supports the basic computational model and checks whether properties hold for a model or not, in the latter case providing a counterexample. In Event-B [2], safety properties are expressed in first-order logic, while constants, data structures, variables and their updates are modelled in a guarded command language. Event-B has a theorem prover tool, the Eclipse-based Rodin platform[2], that supports the basic modelling and analysis, based on generating and discharging proof obligations. Modularity is addressed via refinement: a model is initially abstract and details are added to it in proof-safe manner. Liveness properties are modelled logically or with specific update types.

Contributions. After modelling and analysing OLSR with both Uppaal and Event-B, we found that both formal methods are useful, but at different scales and for emphasising different aspects of modelling and analysis. In this paper, our contribution is to provide a comparison of our respective models as well as of these formal methods, with suggestions for modellers as to when to use one or another. We take into account four main criteria w.r.t. our models (Uppaal and Event-B models) comparison: parts of the protocols that have been modelled, particular properties that have been verified, networks topologies that have been modelled and data structures that have been used when modelling. To overview the applicability of Uppaal and Event-B, we provide a comparison between them by focusing on their specification languages, their mechanism for variable updating, their modularity methods, their verification strategies, their scalability potentials and their real-time modelling capabilities. Based on our

[1] http://uppaal.org/.
[2] http://www.event-b.org/.

considerations, we provide several guidelines for when to use Uppaal or Event-B for formal modelling and analysis.

Outline. We proceed as follows.[3] In Sect. 2 we describe in some detail the formal tools employed in the paper, namely Uppaal and Event-B. In Sect. 3 we overview the OLSR protocol and in Sect. 4 we summarise our modelling of OLSR in Uppaal and Event-B, respectively. In Sect. 5 we compare our Uppaal and Event-B models as well as the frameworks themselves. We draw some usage guidelines of these formal tools in practical situations in Sect. 6.

2 Formal Methods, Model Checking, and Theorem Proving

A formal method usually refers to a framework allowing one to model, analyse, verify, and animate a system. A formal methods has a formal semantics based on mathematics, and can thus provide precise answers to questions about systems properties. A formal method includes a specification (or modelling) language, analysis methods, various modularity mechanisms addressing the scale of a system; nowadays, successful formal methods also have tools associated to them, including editors, analysers, animators, and more.

When modelling the dynamic behaviour of a system with a formal method, each execution step in the model follows from a semantical rule of inference and hence can be checked by a mechanical process. The advantage of formal methods is that they provide valuable means to symbolically examine the entire state space of a system model and establish a correctness or safety property that is true for all possible inputs. These methods have a great potential on improving the correctness and precision of design and development, as they produce reliable results. However, this is rarely done in practice today, except for safety critical systems. In the rather recent past, one of the reasons was the lack of user friendly and scaling tools, combined with the enormous complexity of real systems. Nowadays however, we have good tools for several formal methods, so one of the questions remaining for the adoption of formal methods in industry remains: which tool is more suitable for a certain (type of) system?

In this paper we set out to examine two different tools associated to two formal methods, namely model checking and theorem proving.

2.1 Model Checking–Uppaal's Timed Automata

Model checking (e.g. [9]) is an algorithmic and automatic approach used to validate and verify key correctness properties in finite representations of a formal system model. By modelling the behaviour of a system in mathematical language, model checking exhaustively and automatically checks whether the model meets

[3] The detailed descriptions of our models appear in [16] for the Uppaal model and in [19] for the Event-B model.

a given specification. In model checking, Temporal Logic (TL) is used to specify and check the correct behaviour of a system. One of the most used model checking tools nowadays is the Uppaal model checker.

Uppaal [7,20] is an integrated model checker for modelling, simulating (validating) and verifying real-time systems. It is appropriate for systems that can be modelled as networks of timed automata extended with bounded integer variables, structured data types, functions and synchronisation channels. A timed-automata is a finite-state machine with clock variables that measure time progression. Each automaton can be represented as a graph consists of locations (optionally also consisting invariants) and edges between those locations having guards, synchronisation channels, and updates of some variables. A state of a system is defined by automata's locations, value of clocks, and the value of all local and global variables. An edge can be fired in an automaton which leads to a new state. This edge can be fired separately in the automaton or between different automata used for synchronisation.

Uppaal's verifier uses Computation Tree Logic (CTL) (e.g. [11]) to express system requirements (properties) offering two types of formulas: state formulas and path formulas. State formulas describe individual states of the model, whereas path formulas quantify over paths in the model.

2.2 Theorem Proving–Event-B

Event-B [2] is a formal technique based on the B-Method [1] and on the Action Systems [5] framework, provides means to model and analyse parallel, reactive and distributed systems. Rodin Platform [3] provides automated support for modelling and verifying such systems. Event-B uses two modules for defining system specifications and for expressing system properties, namely `context` and `machine`. A context consists of carrier sets and constants, and their properties are defined as axioms of the model. So, a context deals with the static part of the system whereas a machine contains the dynamic part of the system. A machine can access the contents of a context which is defined by the keyword `Sees` determining the relationship between the machine and the context.

A machine expresses the model state using `variables` that are updated by `events`. Events can have `guards` that need to evaluate to `true`, allowing the event to be executed. When having several events enabled simultaneously, one event is selected non-deterministically. A machine may also contain `invariants`, i.e., properties which must hold for any reachable state in the model. In other words, invariants must be satisfied before and after the occurrence of all events.

The `refinement` is the main developing strategy in Event-B where a machine, let's say machine `A`, is refined by another machine, let's say machine `B`, i.e., $\mathtt{A} \sqsubseteq \mathtt{B}$. This happens when `A`'s behaviour is not altered by `B` in any way and more new variables are added in `B` as well as new events to update the new variables. This type of refinement employed for our modelling is called `superposition` refinement. In order to prove that machine `B` is the refinement of machine `A`, a set of so-called `proof obligations` is generated by the Rodin platform. Some of

these proof obligations are discharged automatically by Rodin and some require interactive discharging with the help of the modeller.

3 An Overview of Optimised Link State Routing

The Optimised Link State Routing (OLSR) is a proactive routing protocol developed for Mobile Ad-hoc Networks (MANETs) and Wireless Mesh Networks (WMNs). OLSR operates as a routing table-driven protocol; each node keeps information about all the other nodes of the network in order to transfer data packets from a source node to a destination node. Examples of information stored in the routing table of a node a are: to get to node b (from a) the next node to take is node c; or, to get to node b from a takes n hops, where a, b, c are nodes in the network and n is a natural number. Keeping the information in the routing table up-to-date is realised by nodes periodically exchanging specific *control* messages. OLSR is an optimisation over other link state protocols, since it decreases the network traffic by restricting the broadcasting of control messages to only specific nodes.

OLSR works in a completely distributed manner and does not require any central entity for coordination. Each node selects a set of one-hop neighbour nodes that have links to the two-hop neighbours of that selector node. The selected nodes are called `MultiPoint Relays (MPRs)` and are allowed to transmit control messages intended for diffusion into the entire network. There are two types of control messages, namely `HELLO` and `TC (Topology Control)` messages.

`HELLO` messages are broadcast every 2 s and are used to determine one-hop and two-hop neighbours of each node as well as to select `MPR` nodes. These messages are only broadcast on single hops (to one-hop neighbours) and are not forwarded. `TC` messages are broadcast every 5 s for building and refreshing topological information in the routing tables. These messages are broadcast on single hops and can be forwarded through the network via `MPR` nodes. Upon receipt of `HELLO` or `TC` messages, the receiving node updates its routing table based on the information in the received control message. Therefore, the topological information is always kept up-to-date in the routing tables in order to deliver data packets to arbitrary destination nodes.

4 Formal Modelling of the OLSR

We now present the overview of our OLSR models, i.e., Uppaal and Event-B models of the OLSR protocol. Both formal models are described in detail in our technical reports [16,19].

4.1 Uppaal Model of the OLSR

In [15], we modelled OLSR in Uppaal as a parallel composition of identical processes, each indicating the behaviour of each node of the network. Every

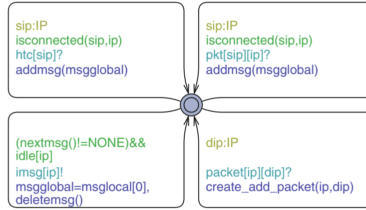

Fig. 1. Overview of model development

process is itself a parallel composition of two timed-automata, i.e., `OLSR` and `Queue`. The `OLSR` automaton is modelling the complete behaviour of the routing protocol [10] and `Queue` automaton (depicted in Fig. 1) is chosen to model the input buffer of every node in the network.

Nodes are able to broadcast and handle different types of messages (`HELLO`, `TC` and `PACKET`) in the network (modelled by `OLSR`) and the connected neighbour nodes can receive the incoming messages and store these messages in their input buffer (modelled by `Queue`). Whenever the `OLSR` is ready to handle a message (is not busy) and there are messages stored in the `Queue`, the `OLSR` and the `Queue` synchronise together on the `imsg` channel, moving a message from the `Queue` to the `OLSR` for processing.

The `OLSR` models the routing table of a node using a local data structure. Routing tables provide all the necessary information to route data packets to different destination nodes. Connectivity between two nodes is modelled by the predicate `isconnected[i][j]`, denoting a node-to-node communication. If two nodes are in transmission range of each other, they can communicate with each other via channels. In order to model rigorous timing behaviour, we defined several clocks for each `OLSR` to model on-time broadcasting control messages, to consider time spent to send every message, and to update and refresh the information in the routing tables.

Based on [10], each node in the network broadcasts a `HELLO` message every 2 s containing the information about the originator of the message and the one-hop neighbours of the `HELLO` message originator. Upon receipt of a `HELLO`, the receiving node updates its routing table for the `HELLO` message originator and its two-hop neighbours (one-hop neighbours of the `HELLO` message originator). The receiving node also selects its `MPR` nodes which are able to broadcast `TC` messages through the network. Such nodes (`MPR`s) then broadcast `TC` messages every 5 s through the network. `TC` messages contain the information about the originator of the `TC` messages, `MPR` nodes of the message originator, etc. When a node receives a `TC` message, it first checks if the message is considered for processing following some conditions. If so, then the receiving node updates its routing table for the `TC` message originator and the `MPR` nodes of the `TC` originator. Afterwards, if the receiving node is an `MPR` and the `TC` message is considered for forwarding, the `TC` is forwarded to the next nodes.

The `Queue` (Fig. 1) models storing incoming messages from other nodes (directly connected neighbour nodes) of the network. The incoming messages are buffered and in turn are sent to the `OLSR` for further processing. Messages can be received only if the receiving node is connected to the sender of the message. In this case, the `Queue` of the receiving nodes stores the messages to its local data queue.

4.2 Event-B Model of the OLSR

In [18], we developed a formal model of the OLSR protocol at five different levels of abstraction (depicted in Fig. 2) using Event-B (Rodin platform). We have defined two contexts containing constants and carrier sets, whose properties are expressed as a list of axioms for the model. These contexts contain the static part of the system. The dynamic part of our system is modelled using five machines that describe the state of the model with their variables which are updated by events. These five machines are related to the contexts and can access them using the keyword sees as shown in Fig. 2. Also, the more abstract machines and contexts are refined into more concrete machines and contexts using keywords 'refines' and 'extends', respectively.

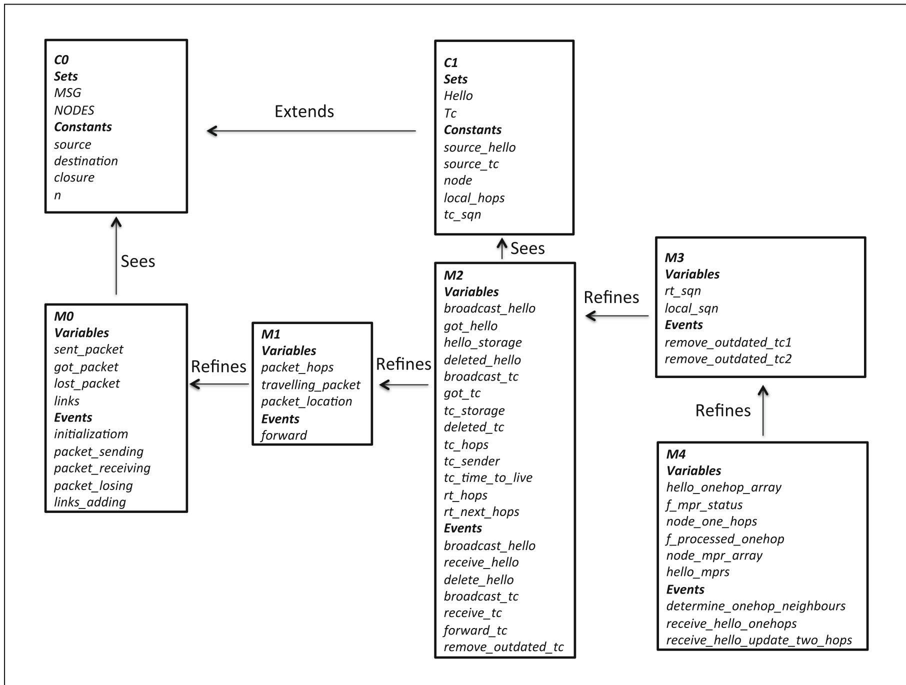

Fig. 2. Overview of model development

Our initial model M0 deals with basic protocol behaviour, i.e., sending, receiving, and losing data packets as well as an abstraction of proactive routing behaviour (adding links between nodes). First refinement M1 models a storing and forwarding architecture when data packets are transferred hop by hop from a source node to a destination node. Second refinement M2 models the basic behaviour of the route discovery protocol, describing the OLSR behaviour when sending and receiving control messages as well as updating routing tables. Third refinement M3 models how the protocol decides to process only new control messages and how to avoid processing control messages with old information. Fourth refinement M4 models the selection of MPR nodes, helping to decrease the traffic in the network.

```
Event  packet_receiving ≙
any
        msg
where
        grd1 : msg ∈ sent_packet \ (got_packet ∪ lost_packet)
then
        act1 : got_packet := got_packet ∪ {msg}
end
```

In M0, data packets are received from a source node to a destination node in an atomic step which is of course not the case in reality. In real protocols, data packets are forwarded hop by hop from a source node to a destination node using multi-hop communication that is modelled in the more concrete machine M1. For instance, event *packet_receiving* models the successful receiving of the data packet *msg* by a destination node. The guard of this event (*grd1*) models that *msg* has not been received or lost yet. When the packet is received, it will be added in the got_packet set.

In M1, the storing and forwarding architecture of data packets is modelled while all nodes are not connected and the data packets must be forwarded hop by hop through the destination. In this step, we model a local storage for each node to store these incoming packets and forward these data packets to next nodes along the path to the destination node.

In M2, nodes are able to broadcast and handle different types of messages (HELLO, TC and PACKET) in the network (modelled by several events). Also routing tables of nodes are modelled as variables, providing the information to deliver data packets to different destination nodes. Every node broadcasts a HELLO message having the information only about the HELLO message originator. Upon receipt of a HELLO message, the corresponding routing table for the originator of the HELLO message is updated. Also, each node broadcasts a TC message containing the information about the TC message originator, number of hops of the TC message, sender of the TC message and time to live of the TC message (number of hops that a TC message can be forwarded). Upon receipt of a TC message, the corresponding routing table for the originator of the TC message is updated and if the TC message is considered for forwarding, it is forwarded to the next nodes.

In M3, we extend the routing table of every node and also add a new variable in the TC message in order to model sequence numbers. Sequence numbers are embedded in TC messages to avoid processing messages with old information.

Also, we defined several events to update the local sequence number of each node and to remove out-dated messages from the network.

In `M4`, we restrict the broadcasting of `TC` messages to only specific nodes, namely `MPR`s, and not all nodes broadcast `TC` messages through the network. We added one-hop neighbours of the `HELLO` message originator in the `HELLO` messages so that upon receipt of a `HELLO` message, the two-hop neighbours of the receiving nodes can be also updated. In this case, nodes can determine their `MPR` nodes and also nodes are able to recognise whether or not they are `MPR` nodes of some other nodes in the network. If some nodes are selected to be `MPR`s, then they can broadcast/forward `TC` messages through the network.

5 Comparison

In this section, we compare our OLSR models, the Uppaal model [15] and the Event-B model [18] as well as the modelling tools Uppaal [7] and Event-B [2].

5.1 Uppaal Model vs Event-B Model

Table 1 depicts an overview of our comparison. We take into the account four main criteria: what parts of the protocol we've modelled, what properties we've verified for our models, for what types of network topologies we modelled the protocol and what data structures we've used.

Table 1. Overview of our models comparison

	Uppaal model	Event-B model
Protocol	Core functionality	Core functionality with timing abstraction
Properties	Route establishment packet delivery non-optimal route finding recovery time	Route establishment packet delivery non-optimal route finding
Topologies	All topologies up to 5 nodes	All topologies with n nodes
Data Structures	Queues	Relations, functions

Protocol. We were able to model the core functionality of the OLSR protocol [10] in both Uppaal and Event-B. This functionality refers to the behaviour that is always required for the protocol to perform. The only feature that we abstracted away in our Event-B model was the timing of messages. In the OLSR protocol [10], `HELLO` and `TC` messages are sent periodically. We have abstracted away the treatment of time in Event-B as this is still incipient, involving a rather different perspective of treating variables as continuous functions of time [4,6].

Table 2. Overview of Uppaal and Event-B comparison

	Uppaal	Event-B
Specification Language	Timed automata, C-like language	Set theory, guarded commands language
Variables Update	Transition: selection guard update	Event: parameter guard action
Modularity	Divided into several automata at the same level of abstraction	Divided into several machines at different levels of abstraction
Verification	CTL automatically providing counterexamples	First-order logic automatically and interactively no counterexamples
Scalability	Small-scale systems (finite)	Large-scale systems (infinite)
Real Time	Precisely models timing variables	Partially models timing variables

Properties. We verified our OLSR model in Uppaal for the following properties: route establishment, packet delivery, optimal route finding, and recovery time. We were able to verify that all nodes in the network can establish routes to different destination nodes as well as deliver data packets to these destinations. We proved by finding a counterexample that OLSR is not always able to find optimal routes to all the destinations as well as showed that OLSR needs a relatively long time to recover after a link breakage in the network [15]. In our Event-B model, we verified our OLSR model for the following properties: route establishment, packet delivery and optimal route finding. We came to the same conclusions as for our Uppaal model. Routes are established to all destinations and data packets are delivered to these destinations; however, these routes may be non-optimal w.r.t. the hop counts. Since we abstracted away from timing properties, we did not investigate the recovery time of OLSR in Event-B.

Topologies. We verified our Uppaal model of OLSR for all network topologies up to 5 nodes. Since the model checking technique suffers from the state space explosion problem, we were not able to extend our analysis for more realistic networks. However, when modelling in Event-B, we were not restricted by the number of nodes in the network and we could verify the protocol for arbitrary networks with n number of nodes.

Data Structures. We modelled the OLSR protocol in Uppaal and Event-B with different data structures. In our Uppaal model, we have defined the `Queue` timed automata to store different types of incoming messages to a node. In Event-B, we modelled the storing architecture using relations between nodes and messages. We defined a specific data structure in Uppaal to model the routing tables,

whereas in Event-B we defined different variables to model routing tables. The types of nodes in the network were defined by integers in Uppaal, while in Event-B we introduced a carrier set to model the network nodes. We defined a common data structure for all types of messages in Uppaal; in Event-B, we introduced different carrier sets for each type of message. We note here that we can have the same data structure (modelling all types of messages, i.e., data packets and control messages) also in Event-B, and this is part of some future generalisation that we plan for modelling various network protocols in Event-B.

5.2 Uppaal vs Event-B

Table 2 depicts an overview of the comparison between Uppaal and Event-B. We detail this table below, namely we compare the specification languages, the variable updating mechanisms, the modularity methods, the verification strategies, the scalability potential, and the real-time modelling capabilities.

Specification Language. The Uppaal model checker uses *timed-automata* as the specification language whereas Event-B is based on *set theory*. In Uppaal, constants, data structures and procedures are defined in a C-like language. In Event-B, constants, data structures, variables and their updates are modelled in a guarded command language.

Variable Updating Mechanism. In Uppaal *transitions* are used to update the variables while in Event-B *events* accomplish the same thing. In both formal methods, the state of the model is determined by the values of the variables. We show the similarities between transitions in Uppaal and events in Event-B by sketching an example based on our models when a node receives a message as depicted in Fig. 3. By this, we also demonstrate how our models in Uppaal and Event-B are equivalent. These similarities are as following:

- *Selection of Parameters.* In Uppaal, the `select` label of a transition consists of a list of `name:type` expressions, where `name` is the variable's name and `type` is its type. As depicted in Fig. 3 (Transition), `IP` is the type of variable `ip`, i.e., an integer in our model. This variable is only accessible for the respective

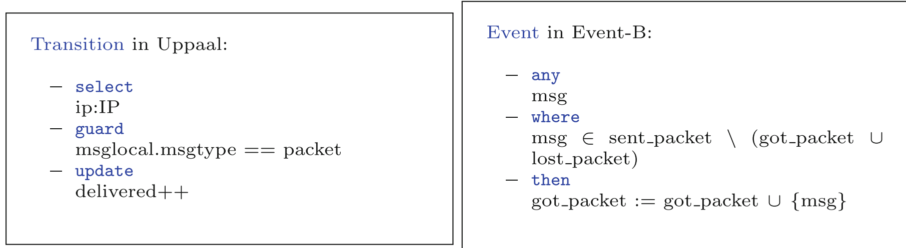

Fig. 3. Transition and event in Uppaal and Event-B.

transition and it takes a non-deterministic value in the range of its respective types (integer type in our model). In Event-B (Event), the `any` clause of an event lists the parameters (or local variables) of the event, i.e., `msg` in Fig. 3; the types of these parameters are usually specified in the guards of the events.

- *Guards.* In Uppaal, the `guard` label refers to logical expressions that determine if the respective transitions are enabled (when guards hold). In Fig. 3 (Transition), `msglocal.msgtype == packet` is the guard of the transition and shows if the received message is a new packet. In Event-B, the `where` clause contains the guards of the events, i.e., the logical conditions for the event to be enabled (when guards hold). The guard of the (Event) in Fig. 3 is shown as $\texttt{msg} \in \texttt{sent_packet} \setminus (\texttt{got_packet} \cup \texttt{lost_packet})$.
- *Updates and Actions.* In Uppaal, the `update` label of a transition contains a list of expressions that update the values of variables. In Fig. 3 (Transition), `delivered++` is the update that increases the value of integer variable `delivered` showing that the packet has been received. In Event-B, the `then` clause lists the actions of the event that modify some variables of the model. In Fig. 3 (Event), $\texttt{got_packet := got_packet} \cup \{\texttt{msg}\}$ is the action that adds the receiving packet to the received messages set. In both frameworks, the variable updating mechanism takes place only if the guards of transitions or events respectively hold.

Modularity. In order to model the whole system's behaviour in Uppaal, several automata are introduced, each modelling different parts of the system. These automata need to synchronise with each other, to keep the consistency and relevance between different parts of the system model. However, it is not always possible to split the system into different automata and thus a system model may remain too complex to understand, having too numerous transitions. In Event-B, different machines are introduced to fully model the behaviour of the system at different levels of abstraction, starting from a very simple and abstract level. This abstract model is stepwise developed using refinement methods to finally model the complete behaviour of the entire system. Consistency between the different levels of refinements is verified by discharging proof obligations. The stepwise development allows to split the complexity of the system into different levels and makes it easier to understand the model and discharge the proof obligations.

Verification. In Uppaal, the required properties are expressed in Computational Tree Logic (CTL) syntax and the whole system model is verified for the defined properties. In Event-B, invariants are used to formulate system properties using first-order logic; the invariants have to be checked for the whole system in order to show the consistency between different levels of abstractions. Properties in Uppaal are discharged fully automatically whereas in Event-B some of the properties are discharged automatically and some are discharged interactively. Uppaal provides counterexamples if a property does not hold; this helps in finding errors in the system. In Event-B, if a proof obligation is not discharged automatically, this typically signals some modelling problem and the modeller is prompted back to remodel certain aspects.

Scalability. Uppaal, like all model checking tools, suffers from the state space explosion problem, hence it is not able to verify very large and complex systems. Event-B allows to verify even large and complex systems. Event-B checks the general validity of a property for *all* models (i.e., also for infinite models) whereas Uppaal is dedicated to small-scale, finite systems.

Real-Time. Uppaal provides clock variables to model timing behaviour of real-time systems whereas for Event-B modelling timing behaviour is still incipient. In Uppaal, clock variables model discrete timing behaviour. In Event-B, advances are made to model hybrid behaviour including discrete and continuous time modelling [4,6], but these are not implemented in the Rodin platform yet. In Event-B, the time can be defined as a function that can be mapped to an integer variable increasing by the events.

6 Conclusions and Usage Guidelines

To resume our experiences of modelling OLSR with Uppaal and Event-B, we essentially found that the two formalisms require different approaches to modelling. In Uppaal, the modeller attempts to capture the whole system, in all its complexity, from the beginning, aided in this task by the modularity technique of splitting the model into communicating time automata. In Event-B, the modeller gets to understand the system's complexity by modelling it in increasingly more detailed levels of abstraction. When we have a conceptually complex system (behaviour of routing protocols), choosing Uppaal or Event-B for modelling it and analysing it is ultimately a matter depending on the modeller's experience.

One can specify properties to prove in both formalisms, but the verification of these properties differs in the two frameworks. In Uppaal, the verification depends on the size of the model and may be unsuccessful if the size is bigger (networks of realistic size) than some arbitrary and typically small value. This is because model checking enforces a brute force verification of properties in all possible states of the system, thus leading relatively fast to overflow. Approaches are taken to overcome this problem, such as partial order reduction techniques [21] and statistical model checking. The former assumes that not all states are worth verifying, and thus defines a priority-based order relation that imposes the verification of the most important states only. The latter employs probabilities and gives results such as the property holds with a 0.99 probability; these probabilities are calculated based on many random walks through the state space (simulations of) the system. In Event-B, the verification of properties is based on logic and proof engines that are built to work for any defined mathematical concepts, including infinite-sized models. When properties are not verified automatically, Uppaal provides counterexamples exposing the offending state: this can be quite useful for correcting errors. In the same situation, the Rodin platform shows the unsatisfied proof obligation and thus the modeller gets some feedback on what does not work. We note here that, if there are flaws in the system, often they are exposed even for small-scale models, see [12,15].

Both Uppaal and Event-B are supported by performant software platforms for modelling and proving; depending on how advanced these platforms are, some aspects can be modelled or not, such as real-time properties. Uppaal was designed to include clock variables and time modelling, while Event-B was designed as a general refinement-based framework. We can precisely model real-time properties of communication protocols in Uppaal, e.g., broadcasting a control message at a certain time. Recently, several approaches were proposed on how to add real-time modelling in Event-B in a conservative manner, e.g. Hybrid Event-B [6] or [4]. This would imply that all variables except clocks are functions of time, so a slight change of perspective is needed here. Real-time properties are typically closely related to implementation details, for instance, to various network parameters; hence, even if we can model timing, when translating the final model into a software product, we might need to alter various properties and parameters anyway.

For modelling and verifying routing protocols, Uppaal remains very useful, as it provides synchronisation mechanisms used in wireless networks: broadcast and binary synchronisation. This allows to closely understand the communication between network nodes. Besides these clear differences, we found that modelling in either framework is quite natural and rewarding and, once the modeller is experienced enough with the framework, quite efficient as well.

To the best of our knowledge, this is the first paper comparing Uppaal and Event-B with respect to what each can model and prove. Relations between model checking and theorem proving in general have been studied before, e.g. [13], where for solving a (rather simple) puzzle, arguments are given for using model checking instead of theorem proving. We note that real systems are very complex nowadays and thus, proving properties for the system, independently of its size, is quite important. Another interesting observation made in [13] is that theorem proving helps in constructing the model, while model checking can be used when we already understand the model quite well. Other approaches connecting model checking and theorem proving are [8], where the idea is to combine the two methods and more recently [24], where refinement is studied in the context of both Uppaal and Event-B.

References

1. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, New York (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6), 447–466 (2010)
4. Abrial, J.-R., Su, W., Zhu, H.: Formalizing hybrid systems with Event-B. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 178–193. Springer, Heidelberg (2012). doi:10.1007/978-3-642-30885-7_13

5. Back, R.J.R., Sere, K.: From action systems to modular systems. In: Naftalin, M., Denvir, T., Bertran, M. (eds.) FME 1994. LNCS, vol. 873, pp. 1–25. Springer, Heidelberg (1994). doi:10.1007/3-540-58555-9_83
6. Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H.: Core hybrid Event-B I: single hybrid Event-B machines. Sci. Comput. Program. **105**, 92–123 (2015)
7. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30080-9_7
8. Berezin, S.: Model checking and theorem proving: a unified framework. Ph.D. thesis, Carnegie Mellon University (2002)
9. Clarke, E.M., Emerson, E.A., Sifakis, J.: Model checking: algorithmic verification and debugging. Commun. ACM **52**(11), 74–84 (2009)
10. Clausen, T., Jacquet, P.: Optimized link state routing protocol (OLSR). RFC 3626 (Experimental) (2003). http://www.ietf.org/rfc/rfc3626
11. Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science, vol. B. Formal Models and Semantics, pp. 995–1072. MIT (1995)
12. Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: Modelling and analysis of AODV in UPPAAL. In: 1st International Workshop on Rigorous Protocol Engineering, pp. 1–6 (2011)
13. Halpern, J., Vardi, M.: Model checking vs. theorem proving: a manifesto. In: Lifschitz, V. (ed.) Artificial Intelligence and Mathematical Theory of Computation, pp. 151–176. Academic Press Professional, Inc. (1991)
14. Johnson, D., Hu, Y., Maltz, D.: The Dynamic Source Routing Protocol (DSR). RFC 4728 (Experimental) (2007). http://www.ietf.org/rfc/rfc4728
15. Kamali, M., Höfner, P., Kamali, M., Petre, L.: Formal analysis of proactive, distributed routing. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 175–189. Springer, Cham (2015). doi:10.1007/978-3-319-22969-0_13
16. Kamali, M., Kamali, M., Petre, L.: Formally analyzing proactive, distributed routing. Technical report 1125, TUCS - Turku Centre for Computer Science (2014)
17. Kamali, M., Petre, L.: Improved recovery for proactive, distributed routing. In: 20th International Conference on Engineering of Complex Computer Systems (ICECCS 2015), pp. 178–181. IEEE (2015)
18. Kamali, M., Petre, L.: Modelling link state routing in Event-B. In: 21st International Conference on Engineering of Complex Computer Systems, ICECCS 2016, pp. 207–210. IEEE (2016)
19. Kamali, M., Petre, L.: Modelling link state routing in Event-B. Technical report 1154, TUCS - Turku Centre for Computer Science (2016)
20. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. Int. J. Softw. Tools Technol. Transf. (STTT) **1**(1), 134–152 (1997)
21. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Compact data structures and state-space reduction for model-checking real-time systems. Real-Time Syst. **25**(2–3), 255–275 (2003)
22. Neumann, A., Aichele, C., Lindner, M.: Better Approach To Mobile Ad-hoc Networking Routing Protocol (B.A.T.M.A.N.). IETF Draft (2008). https://tools.ietf.org/id/draft-openmesh-b-a-t-m-a-n-00.txt
23. Perkins, C., Belding-Royer, E., Das, S.: Ad hoc On-Demand Distance Vector Routing Protocol (AODV). RFC 3561 (Experimental) (2003). http://www.ietf.org/rfc/rfc3561
24. Vain, J., Tsiopoulos, L., Bostrom, P.: Integrating refinement-based methods for developing timed systems. In: Petre, L., Sekerinski, E. (eds.) From Action Systems to Distributed Systems: The Refinement Approach, pp. 171–185. CRC Press (2016)