Exploiting Model Variability in *ABS* to Verify Distributed Algorithms

Wolfgang Leister
Norsk Regnesentral
Oslo, Norway
wolfgang.leister@nr.no

Joakim Bjørk and Rudolf Schlatte and Einar Broch Johnsen
Institute of Informatics, University of Oslo
Oslo, Norway
{joakimbj,rschlatte,einarj}@ifi.uio.no

Andreas Griesmayer
Department of Computing,
Imperial College London, UK
a.griesmayer@imperial.ac.uk

Abstract—We show a way to evaluate functional properties of distributed algorithms by the example of the AODV algorithm in sensor networks, *Creol* and *ABS* models, and component testing. We present a new method to structure the evaluation work into the categories of techniques, perspectives, arrangements, and properties using executable models. We demonstrate how to use this structure for network simulations and component testing using *Creol* models and demonstrate how the delta modelling technique of the *ABS* language can be used to facilitate the approach.

Keywords—formal analysis, modelling, model checking, testing, routing algorithms.

I. Introduction

With increasing miniaturisation of hardware on one hand, and reduced production cost and power consumption on the other, computational devices are becoming virtually omnipresent and pose new challenges in software development. A novel systematic methodology for verification of such distributed system was presented previously [1] on the example of wireless sensor networks (WSN) [2] modelled in the executable modelling language *Creol*. In this paper, we extend this work by giving more details on the verification process and reporting on advancements in modelling by using the Abstract Behavioural Specification (*ABS*), a recently developed successor of the *Creol* modelling language [3].

The sensor network of our case study consists of spatially distributed autonomous sensor nodes that communicate using radio connections. Each node can sense, process, send, and receive data. We concentrate on the verification of a *distributed algorithm* for ad-hoc networks between the sensor nodes to route data packets of the participating nodes. There are many functional and non-functional requirements for WSN: routing must fulfil properties for quality of service (QoS), timing, delay, and network throughput; furthermore, we are interested in properties like mobility and resource consumption. When evaluating WSN, autonomous behaviour of the nodes leads to state space explosion during model checking, making evaluation a complex task that requires a combination of techniques from different verification approaches.

The presented structured methodology to verify distributed algorithms introduces the categories of *techniques*, *perspectives*, *arrangements*, and *properties*. This structure is combined with techniques from simulation, testing, and model checking to create a new, unified method for verification of distributed systems. We demonstrate the approach by evaluating a large set of properties on a network using the *Ad hoc On Demand Distance Vector* (AODV) routing algorithm [4].

We detail the modelling process using the new language *ABS*, a successor to *Creol* with a variety of improved characteristics and features. While the basic structure of the models remains the same, *ABS* allows to employ one single executable model that is suitable for simulation, testing, and model checking without the need to develop separate models for each task. In contrast to the models developed in our previous work [1, 5], the new models employ techniques from *software product line* modelling [6] to structure the executable model with a resulting reduction in model size of more than fifty percent.

The remainder of this paper is organised as follows: After introducing the concept of model variability in *ABS* and presenting the used languages and related work (Section II), we discuss the AODV model developed previously and contrast it with the newly-developed model (Section III). Next we present our categories for the validation process (Section IV), present results from network simulation and component testing (Section V), and conclude in Section VI.

II. Abstract Behavioural Modelling with Variability

Diversity poses a central challenge in modern software development. Typically, engineers create different system variants to address a number of concerns ranging from different application contexts to customer requirements [7]. Model-centric approaches to system development that provide an abstract representation of system structure and behaviour are rapidly gaining popularity. There is a lot of research involving feature description languages [8], architectural languages for components [9], the Unified Modeling Language (UML) [10], and state machine-based notations [11–13]. Development processes such as software product line engineering [6] distinguish between generic artifacts that are common to different system variants and product-level system development; these processes are specifically designed to use (and reuse) high-level artifacts. For this reason, software product line engineering is a promising approach to model system diversity. However, a prerequisite for ensuring the consistency

of different views during software product line engineering is a uniform semantic foundation [7].

In our model of routing and forwarding algorithms in wireless sensor networks, we encountered model variability for a variety of reasons; these include different features of the model (timed vs. untimed, adjustable message loss, different routing protocols and sensor layouts) and code adjustments for testing purposes. In the *Creol* version of the model, a preprocessor-based solution for feature modelling was employed [14] to address model variability; in the *ABS* model, we used a more principled approach based on feature modelling and deltas.

The rest of the section presents in more detail the modelling approach of *Creol* and *ABS* and about product line modelling in general.

A. The Creol language

The previous paper [1] used *Creol* to model the AODV algorithm. *Creol* [15, 16] is an *object-oriented* modelling language that provides an abstract, executable model of the implementation of components. The *Creol* tools are part of the *Credo* tool suite [17] that unifies several simulation and model checking tools. The *Credo* tools support integrated modelling of different aspects of highly re-configurable distributed systems both structural changes of a network and changes in the components and offer formalisms, languages, and tools to describe properties of the model in different levels of detail. These formalisms include various types of automata, procedural, and object-oriented approaches.

To model components, *Creol* provides behavioural interfaces to specify inter-component communication. We use intra-component interfaces together with the behavioural interfaces to derive test specifications to check for conformance between the behavioural model and the *Creol* implementation. Types are separated from classes, and (behavioural) interfaces are used to type objects.

Creol objects can have active behaviour. They are concurrent, so that, conceptually, each object encapsulates its own processor. Each method call in *Creol* results in the creation of a new process. This means that the calling process continues to run and receives the result of the method call later using a *Future variable* [18]. Since all object fields are private, processes running on different objects can run in parallel safely, without the need for locking of data structures.

Scheduling *within an object* is based on explicit processor release points, i.e., processes within an object cooperate on scheduling. This cooperative scheduling makes it not only possible to reason about and prove the correctness of parallel programs (since scheduling points are apparent in the program text), but, in our experience, it makes code also easy to read and understand with confidence.

During object creation, a designated run method is automatically invoked if present in the class definition; this method provides active object behaviour.

Creol includes a compiler, a type-checker, and a simulation platform based on Maude [19], which allow simulation, guided simulation, model testing, and model checking.

TABLE I
ABS LANGUAGE LAYERS AND THEIR ROLE IN MODELLING SYSTEM DIVERSITY.

Language layer	Modelling role
Functional layer	Specifies internal computations in behavioural modules
Concurrent object layer	Specifies communication and synchronisation of behavioural modules
Delta layer	Modifications to core behavioural modules
Product line configuration layer	Links features to sets of delta modules
Product selection layer	Selects features and initialises a product

Creol only provides limited features for expressing diversity through the use of a pre-processor in the extension *CreolE* [14]. This approach has the disadvantage that the model can become hard to read since all possible aspects are present side-to-side, obfuscating the flow of control. Other features not available in *Creol* include user-defined data types and user-defined functions. Hence, a successor to *Creol* was developed to address these deficiencies.

B. Abstract Behavioural Specification

The *Abstract Behavioural Specification* (*ABS*) [3, 20] modelling language and its accompanying tool framework proposes an approach to the engineering of system diversity that provides a uniform semantic foundation. It supports the precise modelling of behaviour for highly configurable, distributed systems in an end-to-end manner. This means that not only the (concurrent) implementation of features is captured, but also the feature space and the dependencies among them. The *ABS* modelling language aims to fill the gap between structural high-level modelling languages, such as UML, and implementation-close formalisms, including programming languages [20]. Furthermore, *ABS* supports the explicit modelling of time-dependent behaviour for object-based systems by means of its real-time extension [21].

ABS has at its core a state-of-the-art, strongly typed, abstract, concurrent, object-based modelling language [3] that is fully executable. *ABS* offers a number of layers to the system engineer. These layers provide a separation of concerns between different aspects of a system model. Table I shows the different language layers provided in *ABS* and their role in the system modelling. We now explain the purpose and particular features of each layer.

The functional layer of ABS is used to provide a model of internal computation that abstracts from low-level implementation details such as the imperative representation of data structures. This layer consists of user-defined parametric algebraic data types and parametric functions over the terms of these types, including pattern matching. The integration of the functional layer into the object-oriented models results in a very succinct representation of internal computation in the objects that allows the engineer to focus on the communication

and diversity aspects in the upper layers of the model without abstracting from data flow in the model. Thus, *ABS* models are abstract, yet faithful to the data and control flow of the target systems. The *ABS* functional layer has no counterpart in the earlier *Creol* language, which relied on a fixed set of data types and had no user-defined functions.

The concurrent object layer of ABS is used to define the modelling artifacts that represent the system entities in terms of concurrent object groups. Since representation objects can be replaced by terms from the algebraic data types of the functional layer, the engineer may abstract from most representation objects in the model. As a consequence, objects in *ABS* are fairly high-level entities, which are more similar to actors [22, 23] than to Java objects. The concurrent object layer is based on asynchronous method calls between concurrent objects, decoupling communication and synchronisation in the models. Shared memory and synchronous method calls are only permitted among closely collaborating synchronous groups of objects in *ABS*. Otherwise, objects communicate asynchronously and use message passing to update the state. Asynchronous method calls do not transfer control between the caller and the callee. Instead, the reply to a method call may be retrieved by synchronising on a *future* [18]. Inside the concurrent object groups, *ABS* uses collaborative scheduling to provide reasoning control in the interleaving of active and reactive behaviour: a method activation can only be suspended by explicitly yielding control. If no method is active, any enabled method activation may proceed. The concurrent object layer is a straightforward extension of the *Creol* object model, keeping most of the semantics and introducing concurrent object groups.

The purpose of the remaining layers in *ABS* is to engineer system diversity. None of these layers are present in *Creol*. While *ABS* is an object-based language and, hence, compatible with the UML world, code reuse by inheritance, which tends to be brittle, is excluded. Instead, system diversity in *ABS* is captured by *delta modelling* (e.g., [24]), which represents a set of systems by a designated core system and a set of system deltas specifying modifications to the core system. Delta modelling is an incremental composition technique for structured diversity that is highly compatible with feature-oriented software development [25] and also a good match for agile and evolutionary development approaches [26].

The delta layer of ABS is used to specify structured changes to the set of classes by adding or removing variables or methods from a class or by redefining methods in the class. A delta consists of a set of such changes, and may additionally add or remove classes. The integration of deltas into a (core) model happens in a given order at compile time, transforming the model. *The product line configuration layer of ABS* is used to define features in the software product line as sets of deltas. Thus, selecting a feature consists of applying the deltas in the corresponding set in a given order. *The product selection layer of ABS* provides the means to configure a system in order to obtain a given variant of the product line. This is done by selecting the features that should be provided in a specific product of the product line.

C. Timed Modelling

To simulate and verify functional correctness of models as well as timing- and performance-related criteria, a notion of time needs to be included in the model. A timed extension for *ABS* has been developed in Bjørk et al. [27]. In its present form, *timed ABS* employs discrete time and run-to-completion semantics, i.e., the maximum amount of computation is performed in each time interval before the (simulated) clock is advanced. Timing behaviour of models is explicitly encoded, thus being visible to the modeller.

The new language elements are:

- A statement `duration(`*best*`, `*worst*`)` that blocks the current object between *best* and *worst* time units where no other process can execute on that object. This can be used to model CPU-intensive tasks.
- A new guard condition: the statement `await duration(`*best*`, `*worst*`)` causes the current process to be suspended between *best* and *worst* time units, letting other processes of the current object execute in the meantime. This can be used to abstractly model the timing behaviour of interactions with external systems, such as interactions with an external database that is not explicitly modelled.
- A function, `now()`, that returns the current model time as a monotonically increasing integer value. In practice, this function can be used for recording completion times and bookkeeping.

Using `duration` and `await duration`, the modeller can express, e.g., message transfer delays in the network and processing delays in the sensor nodes for the AODV model.

D. Related Work

Showing functional correctness and non-functional properties for algorithms employed for WSN helps the developers in their technical choices. Developers use a variety of tools, including measurements on real implementations, simulation, and model-checking. When developing algorithms for packet forwarding in a WSN, simulation results must be compared with the behaviour of known algorithms to get a result approved [28]. Approaches using simulation, testing, and model checking during the development process use one or more of the following: modelling, traces, runtime monitoring by integrating checking software into the code (instrumentation) [29], or generating software from models automatically [30].

Simulation systems are used to analyse performance parameters of communication networks, such as latency, packet loss rate, network throughput, and other metrics. Most of these systems use discrete event simulation. Examples for such simulation systems [31] include OPNET [32], OMNeT++ [33], the network simulator ns-2 [34], its successor ns-3 [35], or mathematical frameworks like MathWorks [36]. Many of these tools have specialised libraries for certain properties, hardware, and network types. While these network simulators are often used to evaluate the performance of algorithms, they are

primarily not designed for model checking tasks. This implies that code for model checking needs to be implemented in these simulators rather than using these features as integrated parts.

The CMC model checker [29] has been applied on existing implementations of AODV by checking an invariant expressing the loop-freeness property. In that work, both specification and implementation errors were found and later corrected in more recent versions of both specification and implementations. CMC interfaces C-programs directly by replacing procedure calls with model-checker code, thus avoiding the need to model AODV. Wibling et al. use the model checking tools SPIN and *UPPAAL* to verify properties for the correct operation of ad hoc routing protocols [37], such as the LUNAR and DSR algorithms [38]. They use *Propagation Localised Broadcasting with Dampening* (PLBD) as a basic operation, and perform model checking on the LUNAR and DSR algorithms. Both LUNAR and DSR are related to AODV, but use different mechanisms. Chiyangwa and Kwiatkowska [39] uncovered in a timing analysis in *UPPAAL* that many AODV connections unnecessarily timed out before a route could be established in large networks. To avoid this, they proposed to set the timeout value dependent of the network diameter.

Timed automata implemented in *UPPAAL* have been used for validating and tuning of temporal configuration parameters and QoS requirements in network models that allow dynamic re-configurations of the network topology by Tschirner et al. [40]. The strength of *UPPAAL* is that both average-case and worst-case behaviours can be analysed. Tschirner et al. compare their results with a similar implementation in OMNeT++. They found that the results from both simulations coincide closely. While the *UPPAAL* implementation is more high-level, the task of implementing the C++ code for OMNeT++ is rather time-consuming.

The model checker *Vereofy* [41, 42], part of the *Credo* tools, was used to analyse aspects of sensor networks and AODV, as presented by Baier et al. [43]. For *Vereofy* formal semantics relies on constraint automata. Thus, a model of a WSN describes the behaviour of the sensor nodes and the network at the interface level. The specification of the interface behaviour of a sensor node is given in terms of *CARML* (Constraint Automata Reactive Module Language) sub-modules for sensing, receiving and sending. For unicast and broadcast the communication media have been modelled as dynamic component connector networks composed with the help of *RSL* (Reo Scripting Language).

While *Vereofy* uses an automaton approach and process algebra with exogenous coordination, *Creol* and *ABS* are based on executable object-oriented models. Note that properties that can be checked by *Creol* or *ABS* are not necessarily suitable to be checked by *Vereofy*, and vice versa. We also used *Vereofy* as a reference for evaluating the properties and as source for the traces employed for the component testing.

Real-Time Maude [44] is a language and tool supporting the formal specification and analysis of real-time and hybrid systems, based on rewriting logic. It is particularly suitable to specify object-oriented real-time systems. Real-Time Maude

can be seen as complementing on the one hand timed/hybrid automaton-based tools such as *UPPAAL*, HyTech, and Kronos, as well as, e.g., timed Petri nets. The OGDC-algorithm used in certain sensor networks has been simulated and model-checked in Real-Time Maude [45]. The comparison of these simulation results in Real-Time Maude against simulation results in ns-2 have uncovered weaknesses in a concrete ns-2 simulation.

III. Modelling the Components and the Routing Algorithm

Distributed applications can be described in terms of components interacting in an open environment based on the mechanisms of *Creol* [46]. This framework models components and the communication between these components, and executes the models in rewriting logic. Different communication patterns, communication properties, and a notion of time are supported. The lower communication layers use tight, loose, and wireless links.

Based on this work, we defined a model of AODV in a WSN using *Creol* [47] that expresses each node and the network as objects with an inner behaviour. The interfaces of the objects describe the communication between the nodes and the network object. In Figure 1, we show the object structure of the model, including the most important interfaces of a node. Note that the object structure shown in Figure 1 is rather generic and can be employed for modelling AODV in other simulators. This interface model is in contrast to the interface model used by the test harness shown later in Figure 4.

Inside a node, its behaviour was implemented in *Creol* as routines that are not unlike real-world implementations. The model contained different aspects (message loss, timed simulation, different routing algorithms) that were enabled or disabled via a pre-processor.

In the remainder of this section we discuss an *ABS* model that has been implemented later with the same interfaces using the experiences from the previous *Creol* model and using the advantages of *ABS*.

The transition from *Creol* to *ABS* reduced the size of the model by about 50%, as measured by line count. Much of the reduction comes from the new language features of *ABS*, predominantly the added functional layer. In addition to being shorter, the new code is also more readable. The model is structured into a basic layer with a number of deltas adding additional features. The basic layer specifies the interfaces and object structures. It implements a simple flooding protocol without routing or retransmission. Based on that simple but working model, a delta replaces the routing functionality in the sensor class with new code that implements the AODV protocol. This layer also adds attributes for storing the local routing tables in each sensor object. Additional deltas implement message timeout and message loss simulation facilities. In the *Creol* model, all these functionalities were implemented in one place and switched on and off by preprocessor directives; the new structure allows to implement, read, and understand the different aspects in isolation.
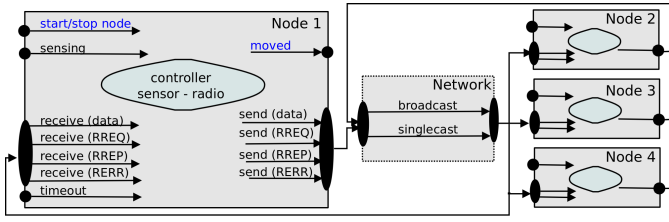
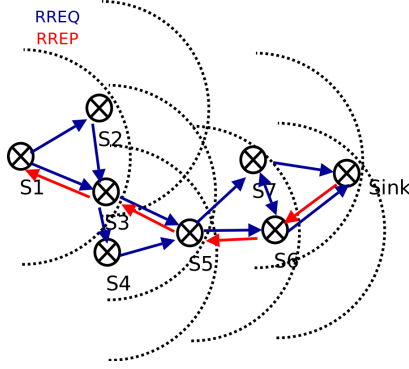Figure 1.   Objects of a WSN model and their communication interfaces.

Figure 2.   Example of AODV message propagation in a WSN with eight nodes.

A. The AODV Routing Algorithm

The purpose of a routing algorithm is to establish a path between a source node and a sink node, so that data can flow from the source node to the sink node via forwarding nodes in-between. AODV is a reactive routing protocol that builds up the entries in the dynamic routing tables of nodes only if needed. AODV can handle network dynamics, e.g., varying wireless link qualities, packet losses, and changing network topologies.

When a node wants to send a message to a sink node and the next hop cannot be retrieved from the local routing table, it initiates a route discovery procedure by broadcasting RREQ (route request) messages. Nodes that receive a RREQ message will either send a RREP (route reply) message to the node that originated the RREQ message if the route is known; otherwise the node will re-broadcast the RREQ message. This procedure continues until the RREQ message reaches a node that has a valid route to the destination node. The RREP message is unicast to the source node through multi-hop communications; as the RREP message propagates, all the intermediate nodes also establish routes to the destination. After the source node has received the RREP message, a route to the destination has been established, and data packages can be sent along this route. As an illustration, Figure 2 shows an example of a small WSN with eight nodes, where the potential RREQ messages are shown in blue, while the RREP messages are shown in red. Note that other paths for the RREP messages are possible in this example.

The essential entries of the routing table in each node include the next hop, a sequence number, and the hop count to the sink node. The hop count is the most common metric for routing to choose between routes when multiple routes exist. The sequence number is a measure of the freshness of a route.

When communication failures imply a broken route, the node that is unable to forward a message will inform other nodes so that the routing tables can be updated. To do this, it sends a RERR (route error) message along the reverse route that is also stored in the nodes. Thus the source node will become aware of the broken route, and initiate a new route discovery procedure.

B. Modelling Message Types

For modelling AODV messages and their operations, we use the functional layer of *ABS*. For simplicity, object references to the sending, origin, and target nodes serve as tokens inside messages. The message payload is represented by a single integer since we do not model higher-level protocols that process message data.

```
data MsgType = RREQ | RREP | RERR | PAYLOAD;
data Message = Msg(MsgType msgType, Node sender,
    Node origin, Node target, Int originSeqNo,
    Int ttl, Int hops, Int content);
```

These data type definitions also generate functions (`msgType()`, `sender()`, etc.) to access the components of a message.

C. Modelling Active Objects

The `Node` type used above is an *interface type*. In *ABS*, classes are not types, so each class must implement at least one interface if it wants to be assignable to a variable. Note that classes without interfaces can be meaningful if they have active behaviour – objects of that class can interact with the rest of the model via references passed to their constructor.

```
interface Node{
    Unit receiveMsg(Message msg);
    Unit timeout(Message msg);
}
interface Sink extends Node {}
interface Sensor extends Node {
    Unit start();
}
```

The methods `receive`, `timeout`, and `start` implement the component behaviour shown in Figure 1. Sending the messages and the network behaviour are implemented by the following interface:

```
interface Network{
    Unit createLink(Node node1, Node node2);
    Unit deleteLink(Node node1, Node node2);
    Unit send(Message msg);
}
```

Besides the method `send()`, the `Network` interface contains the methods `createLink` and `deleteLink` that are used for configuring the network topology and expressing which nodes are neighbours. Neighbours can directly receive each other's messages.

D. Implementing and Augmenting Classes

The class that describes a sensor node is defined as follows:

```
class SensorNode(Network n, Sink s) implements Sensor
{
  ...
  Unit receiveMsg(Message msg) {
    msg = incHopCount(msg);
    this.recordMsg(msg);
    this.forwardMsg(msg);
  }
  ...
}
```

Most of the implementation is elided for brevity, but we show an implementation of `receiveMsg`, which is called by the network when a neighbouring node sends a message. To explain the model variability principles of *ABS*, we first present code that implements a *flooding* protocol where routing is not involved. In this code, the method `forwardMsg` checks whether the incoming message already has been seen; otherwise, it calls *n.send()* for retransmission.

To add the AODV routing protocol to the model, we use a *delta*. This allows us to selectively modify classes, adding and replacing methods and member variables. To add AODV routing, the delta replaces the method `receiveMsg` with a message that also updates the routing table before invoking the original method via the `original()` call.

```
delta AODV {
  modifies class SensorNode{
    modifies Unit receiveMsg(Message msg){
      this.updRoutingTable(msg);
      original();
    }
    ...
  }
}
```

The delta also adds member variables for the routing table, and modifies `forwardMsg` with functionality to handle the different kinds of messages and only re-send messages if the node is on the path to their destination. Again, these parts are elided for brevity.

In the end, when running simulations, we can choose which model to run by defining a product containing the right combination of features, and giving its name as parameter to the *ABS* tool chain.

```
product FloodingBSN(Flooding);
product AODVBSN(AODV);
```

We refer to the work by Schaefer et al. [24] for details about delta modelling.

E. Comparison to the Previous Creol Model

The *ABS* language has been designed from the experiences of *Creol*. We outline some improvements of *ABS*. In *Creol*, only basic data types and classes are available for modelling purposes.

In a first attempt to model AODV in *Creol*, we implemented messages as objects. However, this soon caused an overload of the interpreter in the sense of high execution time and large space requirements, making this model rather impractical. Moreover, since messages do not have an independent behaviour, the use of objects for messages is not required.

Modelling messages as integer numbers would theoretically be possible, but does not allow to add annotations in the form of log messages to the object. Since the underlying Maude interpreter does not allow for writing log files, we need another mechanism for creating traces of messages. From a practical perspective, it is valuable for evaluations to follow the path of messages after having performed a simulation. The final implementation of the *Creol* model, therefore, used maps of strings, where annotations could be added to the message. This allows the extraction of the wanted information from the Maude state file. Modelling messages in this way proved feasible for simulation purposes, but manipulating messages was unwieldy and contributed to the comparatively larger code size of the *Creol* model.

The *Creol* nondeterministic choice operator (`[]`) was used to model message loss. While this was a succinct formulation of the semantics of message loss, it was not really suitable for simulation purposes because we could not adjust the likelihood of message loss. Consequently, *ABS* introduced a proper random function that was used for expressing a parameterisable likelihood of message loss, allowing for Monte Carlo simulation. Despite the use of (pseudo-)random behaviour, simulation results are still reproducible because the seed value for the random number generator in Maude can be supplied as a parameter.

As already mentioned, the *Creol* model mixed different model parameters in-line in the code, relying on a preprocessor to generate the desired code. While this approach works, the resulting model is hard to follow for the reader. The product line- and delta-modelling capabilities of *ABS* proved to be a good approach to extract different behavioural aspects of the model into their own semantic units and generate models with the desired aspects on demand.

IV. METHODOLOGY FOR SIMULATION, COMPONENT TESTING, AND MODEL CHECKING

In this section, we show how to evaluate and validate the functional behaviour of the AODV model in the *Credo* framework [17]. While the *Credo* tool set is based on *Creol*, it is easy to see from the previous discussions that the presented concepts are applicable to the *ABS* model presented in the previous sections. We present the *techniques*, *perspectives*, *arrangements*, and *properties* necessary for the validation and show how to evaluate selected non-functional properties.

A. Techniques for Simulation, Testing, and Model Checking

In order to evaluate the properties of a model, several *techniques* are used to provide the necessary technical measures and procedures to make a model amenable to verification. In general, the following modifications can be applied to the model in preparation for simulation, testing, and model-checking:

Auxiliary variables are added to the model to improve the visibility of a model's behaviour. They must not alter the behaviour and are updated when certain relevant events happen, e.g., a counter is incremented when a new instance is created. When running a simulation, these values can be extracted from the state information and visualised in a step-by-step execution or after the execution of the model execution terminates.

Assertions might be necessary depending on the functional requirements to check. While a number of properties can be checked at the final state using auxiliary variables, properties on the transient behaviour of the model require a check during runtime. For such cases, *Creol* provides *assertions* that stop the execution of a model when the condition is violated. The state that caused the violation of the property is then shown for further analysis.

Monitors are pieces of software that run in parallel to the actual model and are used for properties that go beyond simple assertions. A monitor constitutes an automaton that follows the behaviour of the model to decide the validity of a path.

Guarded execution replaces nondeterministic decisions by calls to a guarding object, here denoted as the *DeuxExMachina* module. This allows to check the behaviour of the model under different conditions, while still maintaining reproducibility of the runs. This technique also specifies certain parameters of the environment, like failure rates of the network.

Fault injection adds a misbehaving node (possibly after a certain time) to check error recovery properties. For instance, misbehaviour in a node may be triggered when energy is used up. Such behaviour can be implemented by sub-classing nodes and implementing certain misbehaving routines in the subclass.

Property search employs model checking techniques to check whether certain conditions hold for all or a given subset of states. Such a search can be directly performed by *Maude*, the execution engine for our interpreter, without the need of implementing the search code in the model.

B. Perspectives

A *perspective* describes the scope of an evaluation. For the AODV model, we developed two perspectives: (*a*) observing the behaviour of the entire network configuration including all nodes and the network and (*b*) observing the behaviour of one node. Testing, simulation, and model checking can be performed from different perspectives and levels of detail for a given model. For AODV, a holistic perspective focuses on the networking aspect of the nodes implementing all the involved nodes and the network in one model. However, for model checking such a model leads to a high number of states and long execution time. Therefore, for realistic models the networking perspective is not feasible.

For the perspective of testing a single node, we use the same model code for the nodes in the holistic perspective, but instantiate only one node explicitly. The network is replaced by a *test harness* that impersonates the network and the remaining nodes. The behaviour and responses of the test harness are

determined by a rule set that is derived from traced messages between the nodes, as outlined in Section V-B.

C. Arrangements

An *arrangement* denotes a set of configuration settings that influences how the model operates. Examples are the use of untimed or timed models, changes to the node topology, perfect or unreliable communication, communication failures, timeouts, and energy consumption. *ABS* supports different arrangements natively using its delta layer, while for *Creol* a preprocessor can be used. Examples for arrangement entities that can be selected in the models, together with implementation details for the AODV model, are given in the following: The *communication behaviour* in our model can be set to be either reliable, non-deterministic, or one of several packet loss patterns including random packet loss. (Note that pure non-deterministic behaviour in a simulation currently is not useful due to restrictions in the implementation of the underlying runtime system, and in general because of non-reproducibility of simulation results.) Using the differences in communication behaviour we can study how the algorithm behaves when communication packets can get lost.

Topology changes are used to check the robustness of the protocol. They can be triggered by certain events, e.g., after a certain number of messages or after a certain amount of time for timed models. A topology change affects the connection matrix in the network and triggers the AODV algorithm to find new routes in the model.

The *timed model* is realized using discrete time steps and introducing a global clock in the network object and internal clocks in the nodes that are synchronised when a task is performed in one or more nodes. This allows, e.g., to reason about messages being sent simultaneously, which eventually will lead to packet loss. Also, the effect of collisions can be shown without using non-deterministic packet loss. The use of a timed model is most viable together with topology changes since the topology needs to be re-installed for a state when another branch is searched in model checking.

Energy consumption is modelled using an auxiliary variable in each sensor node with an initial amount of energy. For each operation, a certain amount of energy is subtracted until the capacity is too low to perform operations on the radio. This indicates a malfunction of the sensor node. Such a node does not perform any actions and represents a topology change of the network, since given path are no longer valid. This allows us to identify in which cases an energy-restricted network can perform communication and whether AODV can find routes around an energy-empty node.

Note that arrangements for memory and buffer sizes can be implemented similarly. When maximum memory size is reached, a node will alter its behaviour and stop performing certain actions.

Timeouts are modelled nondeterministically by the use of a global guarding object and can occur between a message is sent and the corresponding answer is received. AODV employs

timeouts in order to work in environments where communication errors can occur and sends messages repeatedly in case an expected reply has not been received from the network.

D. Properties

A *functional property* is a concrete condition that can be checked for given arrangements, while non-functional properties are values given by metrics. For AODV, we chose the following functional properties: *a*) correct-operation, *b*) loop-freeness, *c*) single-sensor challenge-response properties, *d*) shortest-path, *e*) deadlock-freeness (both for node and for protocol), *f*) miscellaneous composed system properties, and additionally some non-functional properties.

Correct-operation: For a routing algorithm to be correct, it must find a path if a path exists, i.e., it is *valid* for some duration longer than what is required to set up a route from sender to receiver [37, 38]. Checking this property requires the algorithm-independent predicate whether a route exists. In the absence of topology changes, this predicate can be calculated beforehand. When topology changes are possible, however, we need to check the existence of a path between sender and receiver at any step in the algorithm. Since checking this property in *Creol* involves explicitly visiting all nodes, this increases the reachable state-space of the model. To evaluate this predicate effectively, a suitable implementation would be to interface a *Maude* function, which is possible in *ABS*. Meseguer and Rosu [48] give an overview of existing work on model-checking language semantics in Maude, which can be used for *Creol* and *ABS*.

A related property to evaluate is whether a route is re-established after a transmission error given a path still exists. We also evaluate how long the path is interrupted after a transmission error occurs.

Loop-freeness: A routing loop is a situation where the entries in the routing tables form a circular path, thus preventing packets from reaching the destination. The invariant for loop-freeness [29] of AODV must be valid for all nodes. It uses *sequence numbers* of adjacent nodes, and the number of hops in the routing tables as input. The loop-freeness property is checked every time a message is transmitted between nodes. To do this the network-object calls a routine that checks the loop-freeness invariant in an assertion. Since this assertion is complex and contains nested loops, it again should be implemented as a call to a *Maude* function instead of *Creol* or *ABS* code.

Single-sensor challenge-response: The reaction of one node under test is checked using component testing (Section V-B). Messages are sent to the node under test, and the responses from this node are matched against all correct responses. The correct responses are extracted from specifications or from running simulations using different implementations. The single-sensor properties that can be checked express a certain behaviour or the absence of a certain behaviour after a challenge, e.g., whether an incoming message leads to a specified state change in the node or whether the node sends an expected response messages.

Shortest-path: Here, we investigate whether the AODV algorithm finds the shortest path for the paths between the source and sink node; also other metrics for paths could be checked. While AODV finds the shortest path in the case of no packet loss, it does not always fulfil the shortest-path property in the case of packet loss. To check this property we count the number of hops that each payload-message takes from the source to the sink and compare it with the shortest existing path between the source and sink.

Deadlock freedom: Deadlocks in a node, in the protocol or in the model are a threat to robustness, and can reveal errors in the specification, implementation, or model. Global deadlocks will automatically be detected by the underlying Maude implementation and result in an inspectable error state of the model.

Miscellaneous composed-system properties: Examples are properties that state whether valid routes stay valid, avoidance of useless RREQ messages, number of messages received, timing properties, and network connectivity. Most of these are implemented by adding counter variables and predicates.

Non-Functional Properties: Non-functional properties from the application domain such as timing, throughput, delivery ratio, network connectivity, energy consumption, memory and buffer sizes, properties of the wireless channel, interferences, mobility, or other QoS properties can be evaluated by using counter variables and additional code for the model. Note that for most non-functional properties the use of *Creol* might not be the optimal choice, since it is best suited for the evaluation of conformance or violation of non-functional properties. *ABS* has some support for modelling deployment scenarios and resource consumption [49, 50], which could be extended to cater to our modelling requirements.

V. HOLISTIC AND COMPONENT TESTING

Instrumented *Creol* and *ABS* models can be used for different verification and testing techniques: symbolic simulation, guarded test case execution, and model checking. Auxiliary code for assertions and monitor state is added and executed together with the model code. This increases the size of the states and therefore poses a handicap for model checking. A more light weight approach would implement the monitors directly within the checking tools. This is, however, not yet available in the analysis tools. All experiments in this section were first performed on the *Creol* models. TABLE II gives a detailed list of properties that were identified as being of interest for the network example with the symbol ● marking properties that were evaluated, ⊙ partially evaluated and ○ not evaluated yet for simulation and testing. Model checking was used for generating execution traces for testing as detailed in Section V-B. The *Creol* models have been re-implemented in part to *ABS*. Unsurprisingly, these show the same behaviour while the model size, measured in lines of code, is reduced.

A. Holistic Testing

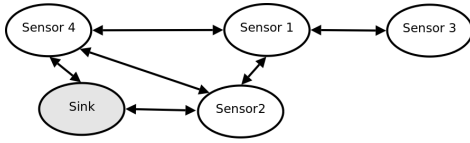For our evaluation of the network properties we used simulation using techniques such as auxiliary variables and

Figure 3. The network used as example in our simulations.

#	t steps	energy	loss	timeout	#rewrites	time
5	500	–	none	never	$9.4 \cdot 10^6$	17.1s
	5000	–	none	never	$62.8 \cdot 10^6$	114.8s
	500	–	10%	never	$10.7 \cdot 10^6$	19.5s
	500	–	10%	1/10	$12.1 \cdot 10^6$	22.3s
	500	50	10%	1/10	$8.3 \cdot 10^6$	15.5s
	untimed	50	10%	1/10	$11.6 \cdot 10^6$	17.9s
	untimed	–	10%	never	$32.5 \cdot 10^6$	14.8s
6	untimed	–	10%	never	$90.5 \cdot 10^6$	40.9s
15	5000	–	none	never	$2.7 \cdot 10^9$	31m
30	5000	–	none	never	$24.8 \cdot 10^9$	8h

assertions. Most of our experiments used a network with symmetrical communication via four sensor nodes and one sink node, as shown in Figure 3. We also experimented with models of 6, 15, and 30 nodes for selected arrangements. The evaluated model consists of code for the network nodes as well as an explicitly modelled network that transmits the messages between the nodes such that, e.g., a broadcast message of *Sensor 2* reaches the nodes *Sink*, *Sensor 1*, and *Sensor 4*. This gives the flexibility to simulate the AODV model using various arrangements including reliable networks, lossy networks, timeouts, energy consumption, and timed modelling.

Symmetrical communication means that whenever a node *A* can transmit a message to node *B*, then communication in the reverse direction can also take place. By changes to the network structure, we also could show that this is a property the AODV routing algorithm in fact relies on. This is because if node *B* receives a broadcast message from *A*, it updates is routing entry for messages destined for *A*. We also checked selected properties from the classes (a), (b), (d), (e), and (f) presented in Section IV-D for the composed network.

Reliable communication: As long as the network is connected, the evaluations showed that the modelled AODV algorithm fulfils the properties (a), (b), (d), (e), and (f) of Section IV-D. We emphasised on the evaluation of packet loss, and loop-freeness assertion. Other predicates for loop-freeness were also used (which failed as expected), and small, faulty changes in the model were introduced (which led to expected failures of the loop-freeness property). The shortest path property was fulfilled in all simulated occasions.

Lossy communication: When simulating lossy communication both for singlecast and for broadcast messages, the packet loss rate f.(xxvi) increases as expected. We also observed an increased number of RREQ and RREP messages in the system using auxiliary variables.

Timeouts: The model allows re-sending of lost RREQ messages up to a certain number of times, using a timeout mechanism. We could observe that this mechanism decreased the packet loss rate, f.(xxvi), but at the same time does not prevent all packet loss for payload packets.

Energy consumption: Using the energy consumption arrangement we can force a communication failure of certain nodes after some actions. Using this arrangement we can study the re-routing behaviour in detail, including the packet loss rate f.(xxvi).

Timed model: Using the timed model we can study the number of time steps needed for sending messages, as well as controlling the number of actions being performed simultaneously. We observed that the packet loss rate f.(xxvi) is different to

the untimed case, which is expected.

Using the timed model, we could observe a model deadlock, e.(xix), which is caused by the way the model is implemented, and certain properties of the current implementation of the *Creol* runtime system. This observation made changes in the model implementation necessary using asynchronous method calls.

The properties f.(xxi), f.(xxiii), and f.(xxiv) could not be evaluated in a satisfactory manner as they require to store all messages during the simulation. Although the properties can be modelled and evaluated in principle, such an arrangement leads to state explosion and exceeds time and memory constraints of our current setting.

The developed *Creol* model was evaluated by using simulation for sample arrangements and properties. The entire model contains about 1600 lines of *Creol* code excluding comments. After compilation, the resulting code size was about 1050 lines of Maude code, depending on the arrangement. We varied the timing behaviour, the energy consumption, the message loss behaviour, and the timeout behaviour of the model as well as the number of nodes. The results for the tested cases considering the number of rewrites, and execution time on an AMD Athlon 64 Dual core processor with 1.8 GHz is shown in TABLE III. The timing behaviour and the number of nodes are the most significant parameters.

While these values may seem high for a simulation system, we emphasise that the purpose of the *Creol* model is to offer one model that is suitable for several perspectives. While the transition from simulation to model checking consists in changing some few Maude statements, the search space during model checking gets combinatorially too high to be viable already for a low number of nodes.

B. Component Testing of One Node

For component testing, we use one node under test with the same code as for holistic testing. However, we replace the network and all the other nodes using a test harness shown in Figure 4. This harness simulates the possible behaviour of the network and the further nodes as visible by the node under test. The output signals of the node under test are connected

| Property	Description	Evaluation	S	T	Property	Description	Evaluation	S	T
a	Correct Operation	yes; for some arrangements.	●		c.(xiii)	updates terminate	yes (implicitly during other tests)		●
b	Loop-Freeness	yes	●		c.(xiv)	update success	yes (implicitly during other tests)		●
c	Sgl-sensor challenge-resp.	yes		●	c.(xv)	only one RREQ	n/a (needs timed interpreter)		
c.(i)	always send with own ID	yes, as invariant during other tests		●	c.(xvi)	Rec. in IDLE mode	n/a		
c.(ii)	msg leads to valid route	yes (inferred from other tests)		●	d	Shortest-Path	yes	●	
c.(iii)	RREQ w/o route⇒RREQ bc.	yes		●	e	Deadlock-Freeness	partially	⊙	
c.(iv)	RREQ for me leads to RREP	yes		●	e.(xvii)	node deadlock	no		
c.(v)	RREP triggers route to originator	yes		●	e.(xviii)	protocol deadlock	yes	●	
c.(vi)	RREP is rebroadcasted	yes		●	e.(xix)	model deadlock	yes	●	
c.(vii)	send if route known	yes (no send if route unknown)		●	f	Misc. Composed-System	yes	●	
c.(viii)	routing table integrity	no		○	f.(xx)	route stays valid	yes	●	
c.(ix)	all msg for sink	yes		●	f.(xxi)	only data msg	possible, not done	○	
c.(x)	processing without receive	yes (during other tests)		●	f.(xxii)	no RERR	yes	●	
c.(xi)	increasing sequence number	yes, as invariant during other tests		●	f.(xxiii)	no useless RREQ	possible, not done	○	
c.(xii)	neighbour update triggers	n/a (not accessible in black-box test)			f.(xxiv)	RREQ triggers RREP	possible, not done	○	
					f.(xxv)	# msg.rec.	yes	●	
					f.(xxvi)	packet loss	yes	●	
					f.(xxvii)	timing properties	partially	⊙	
					f.(xxviii)	network connectivity	yes	●	
					f.(xxix)	parameter tuning	partially	⊙	
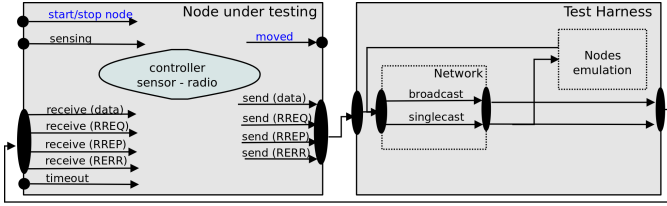
Figure 4. Testing of one node using the network object as a test harness. The test harness replaces the network and all other nodes in the network.

to the test harness, which then in turn generates the inputs that would be received from the real network. The test is then evaluated by studying the output messages of a node when the input messages for the test case are supplied by the harness.

1) Test harness: The task of the test harness is to send messages to the interfaces of the node under test, and to observe its answers. Both input messages and expected answers can be generated from the specification or from traces of real systems or other simulations.

Although incoming broadcast, singlecast, and outgoing packets involve invoking different methods, the *ABS* language, with its object-level parallelism, makes it easy to encode a test case as a single sequential list of statements. Incoming messages are stored in a one-element buffer; the test case simply performs a blocking read on that buffer when waiting for a message from the object under test before sending out the next message to the object. In this way, both creating a test case by hand and generating test cases from recorded traces are feasible.

A test verdict is reached by running the test harness in parallel with the object under test. If the test harness *deadlocks*,

it expects a message from the object under test that is not arriving, a test verdict of *Fail* is reached. The other reason for test failure is an incoming message that does not conform to the expectations of the test harness; e.g. by being of the wrong type or having the wrong content.

A test verdict of *Success* is reached if the test harness completes the test case and the object under test conforms to the tester's expectations in all cases.

2) Traces: In addition to domain-specific single-object properties, test cases can be generated from the model implemented with *Vereofy* [43] (see Section II-D). *Vereofy* is a different language and uses independent mechanisms than both *Creol* and *ABS* to create the traces. Therefore, we can compare the two different models against each other, with the *Vereofy* traces as a partial specification for the behaviour of the *Creol* and *ABS* models.

To receive the traces from *Vereofy*, we collect the exchanged data, the content of all variables in the nodes and buffers in the network, and the output of the automaton before and after each step, in addition to the exchanged data. After the state information is removed, we receive a sequence of messages that are exchanged simultaneously.

The messages in the trace are defined using a *Vereofy* struct:

```
TYPE message_t = struct {
  // determine the type of the message
  message_type_t message_type;
  id_t          dest_id;
  // encapsulation of sender and receiver IDs
  address_t     to_ip;
  id_t          from_ip;
  // case 1: sending AODV messages
  hop_counter_t hop_count;
  seq_no_t      dest_seq_no;
  id_t          orig_id;
  seq_no_t      orig_seq_no;
```

```
Bool           unknown_seq_no;
// omit TTL and XFlag for AODV.
// case 2: for sending data messages
data_type_t    the_data;
};
```

In the following example, Node 1 sends three RREQ messages to find a route to the Sink (Node 0). The RREP generated by the Sink does not arrive, since it remains in a buffer.

```
{send[1]={RREQ,0,2,1,0,0,1,1,1,data0}}
{receive[0]={RREQ,0,2,1,0,0,1,1,1,data0}}
{send[0]={RREP,1,1,0,0,1,0,0,0,data0},
 send[1]={RREQ,0,2,1,0,0,1,2,1,data0}}
{receive[0]={RREQ,0,2,1,0,0,1,2,1,data0}}
{send[1]={RREQ,0,2,1,0,0,1,3,1,data0}}
```

In the tester for Node 1, written in *ABS*, the run-method waits for the messages denoted as `send[1]`, and sends messages denoted as `receive[1]`. A trace can contain messages that are sent simultaneously, such as the third statement of the above trace. Synchronous communication, the calls of `send[0]={RREP,1,1,...}` and `receicve[1]={RREP,1,1,...}`, take place simultaneously.

Traces received from the node under test are tested against *message patterns*, i.e., we remove details that could lead to spurious test failures not expressing a malfunctioning system. For example, the message sequence number can be chosen by the node, the only requirement is that it be monotonically increasing. This property is checked using an invariant in the tester, but a different concrete message number than that used by the *Vereofy* model will not lead to test failure.

C. Other Mechanisms in the Credo Tools

The *Credo* tools based on *Creol*, *Vereofy*, and *UPPAAL* offer different ways of modelling, supporting different techniques, perspectives, arrangements, and evaluation of properties. The evaluations shown in TABLE II have been performed with a similar model in *Vereofy* for a comparison of results where this was suitable [51]. This shows that *Vereofy* is suitable for Properties c, e, and f.(xx) to f.(xxiv). *Vereofy* offers a model-checking approach based on Reo automata using the exogenous coordination model where the components are represented by their behavioural interfaces. Similar to *Creol* and *ABS*, *Vereofy* supports the verification of components and their communication structure. The concrete case study is described elsewhere [43].

For completeness, we mention that selected properties have been evaluated in *UPPAAL*. However, this model only implemented connectivity between sensors rather than ADOV [40]. Properties in the classes f.(xxv) to f.(xxix) have been verified. We mention, however, that *UPPAAL* is capable for other properties, as the work by Chiyangwa and Kwiatkowska [39] and Wibling et al. [37, 38] shows.

VI. CONCLUSION

We presented a structured methodology for the evaluation of complex distributed systems by introducing the dimensions of *techniques*, *perspectives*, *arrangements*, and *properties* for this evaluation. We divided the properties used for this evaluation into six property classes, and performed network simulations of the composed system, and component testing of a single node.

Using the network simulation, we evaluated several arrangements. While most of the properties were fulfilled as expected, some properties did not validate in the simulation; this either due to bugs in the model, artificially introduced misbehaviour in the model, properties of the modelled AODV algorithm, or property variants that are not supposed to validate successfully. In one occasion, we could detect deadlocks in the model in a timed-model arrangement, which could be recognised and fixed afterwards. Evaluating other protocols in sensor networks, e.g., proactive dynamic routing protocols, is possible using the variability features of *ABS*, with test cases and test scenarios adapted to these new protocols.

Using component testing, we validated the correct behaviour of a single node against properties extracted from the specification of the AODV algorithm. No deviations from specified component behaviour were identified in this process, which is unsurprising since components had already been extensively used for simulation and animation during initial model development at that point in time. However, the test suite served as an excellent tool in regression testing during subsequent changes and extensions of the model.

Evaluating the properties of the AODV algorithm, we had previously encountered several challenges, such as modelling suitable abstractions, using language constructs of *Creol*, and observing the properties from a suitable perspective. The major challenge when evaluating the AODV algorithm from a network perspective is to avoid a high number of states in the underlying interpreter. We showed how to overcome modelling difficulties in the existing *Creol* model by refactoring it into a software product line in *ABS*, thereby significantly reducing code size and increasing maintainability and understandability, which was also aided by the higher-level language features of *ABS*.

Besides simulation and state space search using the interpreter, a current line of research concentrates on the automated extraction of a verifiable model for model checking. The approach assumes finite data and a bound on the messages and translates *ABS* models to the input language for MC-MAS [52], a model checker for multi agent systems. While this approach does not allow verification of large systems, it integrates the presented approach by providing new techniques for test case generation and abstraction. A publication on this topic is currently under submission.

The main objective of this study was to evaluate how *Creol* and *ABS* can be applied to complex, distributed algorithms in networks. We found the *Creol* and *ABS* languages and their tools useful in the evaluation of functional properties of the AODV algorithm and we gained insight into how complex algorithms like AODV work. We observed how small changes in the algorithm, and in the chosen arrangement, affect its behaviour so that certain properties fail. We studied these

properties in the implementation of our model, which will lead to further investigation of the reasons for its behaviour or misbehaviour, and the development of better formalisms and languages for modelling and evaluation of properties.

Systems such as *Vereofy* are more specific towards model checking using the automaton approach that is farther from real programs than *ABS* and *Creol*. On the other side, simulators, such as ns-2, ns-3, or OMNeT++, are better suited to evaluate the values of properties, such as timing or power consumption. However, these do not offer native facilities for model checking. Thus, the choice of language and simulating system is highly dependent on the question to a model and a simulation.

The choice of evaluation and simulation tools for distributed algorithms depends on the goal of the evaluation. *ABS* is a further development of *Creol* and contains most of *Creol*'s features that have shown to be useful. *ABS* has mended some of the restrictions of *Creol*. *ABS* and *Creol* belong to the languages that allow executable models that are suited for both simulation, model checking, and testing.

REFERENCES

[1] W. Leister, J. Bjørk, R. Schlatte, and A. Griesmayer, "Verifying distributed algorithms with executable Creol models," *Proc. First International Conference on Performance, Safety and Robustness in Complex Systems and Applications (PESARO'11)*, 2011.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer Networks*, vol. 38, no. 4, pp. 393–422, 2002.

[3] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen, "ABS: A core language for abstract behavioral specification," in *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO'10)*, ser. Lecture Notes in Computer Science, B. Aichernig, F. S. de Boer, and M. M. Bonsangue, Eds., vol. 6957. Springer-Verlag, 2012, pp. 142–164.

[4] C. Perkins, E. Belding-Royer, and S. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing," RFC 3561 (Experimental), Jul. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3561.txt

[5] W. Leister, J. Bjørk, R. Schlatte, and A. Griesmayer, "Validation of creol models for routing algorithms in wireless sensor networks," Norsk Regnesentral, Oslo, Norway, Report 1024, 2010.

[6] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlag, 2005.

[7] I. Schaefer and R. Hähnle, "Formal methods in software product line engineering," *IEEE Computer*, vol. 44, no. 2, pp. 82–85, Feb. 2011.

[8] Q. Boucher, A. Classen, P. Faber, and P. Heymans, "Introducing TVL, a text-based feature modelling language," in *Proc. 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*. University of Duisburg-Essen, Jan 2010, pp. 159–162.

[9] N. Medvidovic and R. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[10] J. E. Rumbaugh, I. Jacobson, and G. Booch, *The unified modeling language reference manual*. Addison-Wesley-Longman, 1999.

[11] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel, *Component-based product line engineering with UML*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[12] H. Gomaa, *Designing Software Product Lines with UML*. Addison Wesley, 2004.

[13] T. Ziadi, L. Hélouët, and J.-M. Jézéquel, "Towards a UML profile for software product lines," in *Revised Papers 5th Workshop on Product Familiy Engineering (PFE'03)*, ser. Lecture Notes in Computer Science, vol. 3014. Springer Berlin / Heidelberg, 2004, pp. 129–139.

[14] W. Leister, "CreolE — a pragmatic extension to Creol," Norsk Regnesentral, Note DART/06/09, August 2009.

[15] E. B. Johnsen and O. Owe, "An asynchronous communication model for distributed concurrent objects," *Software and Systems Modeling*, vol. 6, no. 1, pp. 35–58, 2007.

[16] M. Kyas, *Creol Tools User Guide*, 0.0n ed., Institutt for Informatikk, Universitetet i Oslo, Postboks 1080 Blindern, 0316 Oslo, Norway, May 2009.

[17] I. Grabe, M. M. Jaghoori, B. Aichernig, T. Blechmann, F. de Boer, A. Griesmayer, E. B. Johnsen, J. Klein, S. Klüppelholz, M. Kyas, W. Leister, R. Schlatte, A. Stam, M. Steffen, S. Tschirner, X. Liang, and W. Yi, "Credo methodology. Modeling and analyzing a peer-to-peer system in Credo," in *3rd International Workshop on Harnessing Theories for Tool Support in Software (TTSS'09)*, 2009.

[18] F. S. de Boer, D. Clarke, and E. B. Johnsen, "A complete guide to the future," in *Proc. 16th European Symposium on Programming (ESOP'07)*, ser. Lecture Notes in Computer Science, R. de Nicola, Ed., vol. 4421. Springer-Verlag, Mar. 2007, pp. 316–330.

[19] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, "Maude: Specification and programming in rewriting logic," *Theoretical Computer Science*, vol. 285, no. 2, pp. 187–243, 2002.

[20] D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong, "Modeling spatial and temporal variability with the HATS abstract behavioral modeling language," in *Proc. 11th Intl. School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011)*, ser. Lecture Notes in Computer Science, M. Bernardo and V. Issarny, Eds., vol. 6659. Springer-Verlag, 2011, pp. 417–457.

[21] J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa, "User-defined schedulers for real-time concurrent objects," *Innovations in Systems and Software Engineering*, pp. 1–15, 2012.

[22] G. A. Agha, *ACTORS: A Model of Concurrent Computations in Distributed Systems*. Cambridge, Mass.: The MIT Press, 1986.

[23] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2–3, pp. 202–220, 2009.

[24] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-oriented programming of software product lines," in *Proc. 14th International Conference on Software Product Lines (SPLC'10)*, ser. Lecture Notes in Computer Science, J. Bosch and J. Lee, Eds., vol. 6287. Springer, 2010, pp. 77–91.

[25] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Trans. Software Eng.*, vol. 30, no. 6, 2004.

[26] K. Beck, *Extreme Programming*. Addison-Wesley, 1999.

[27] J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa, "User-defined schedulers for real-time concurrent objects," *Innovations in Systems and Software Engineering*, 2012, to appear.

[28] I. Stojmenovic, "Simulations in wireless sensor and ad hoc networks: matching and advancing models, metrics, and solutions," *IEEE Communications Magazine*, vol. 46, no. 12, pp. 102–107, 2008.

[29] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, "CMC: a pragmatic approach to model checking real code," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 75–88, 2002.

[30] M. M. R. Mozumdar, F. Gregoretti, L. Lavagno, L. Vanzago, and S. Olivieri, "A framework for modeling, simulation and automatic code generation of sensor network application," in *Proc. Fifth Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON'08)*. IEEE, 2008, pp. 515–522.

[31] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Proc. 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops (Simutools'08)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, pp. 60:1–60:10.

[32] OPNET, "Application and network performance," web pages, last accessed February 6, 2011. [Online]. Available: www.opnet.com

[33] OMNeT++, "Bibliography," web pages, last accessed February 6, 2011. [Online]. Available: omnetpp.org/publications

[34] ns-2, "The Network Simulator ns-2," web pages, last accessed February 6, 2011. [Online]. Available: www.isi.edu/nsnam/ns/

[35] ns-3, "Papers," wiki pages, last accessed February 6, 2011. [Online]. Available: www.nsnam.org/wiki/index.php/Papers

[36] MathWorks, "Solutions," web pages, last accessed February 6, 2011. [Online]. Available: www.mathworks.se/solutions

[37] O. Wibling, J. Parrow, and A. N. Pears, "Automatized verification of ad hoc routing protocols," in *Proc. Formal Techniques for Networked and Distributed Systems (FORTE'04)*, ser. Lecture Notes in Computer Science, vol. 3235. Springer, 2004, pp. 343–358.

[38] ——, "Ad hoc routing protocol verification through broadcast abstraction," in *Proc. Formal Techniques for Networked and Distributed Systems (FORTE'05)*, ser. Lecture Notes in Computer Science, vol. 3731. Springer, 2005, pp. 128–142.

[39] S. Chiyangwa and M. Z. Kwiatkowska, "A timing analysis of AODV," in *Proc. Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, ser. Lecture Notes in Computer Science, vol. 3535. Springer, 2005, pp. 306–321.

[40] S. Tschirner, X. Liang, and W. Yi, "Model-based validation of QoS properties of biomedical sensor networks," in *Proc. 7th ACM international conference on Embedded software (EMSOFT'08)*. New York, NY, USA: ACM, 2008, pp. 69–78.

[41] C. Baier, T. Blechmann, J. Klein, and S. Klüppelholz, "Formal verification of components and connectors," in *Proc. 7th Software Technologies Concertation on Formal Methods for Components and Objects (FMCO'08)*, ser. Lecture Notes in Computer Science, vol. 5751. Springer-Verlag, 2009, pp. 82–101.

[42] S. Klüppelholz, "Verification of branching-time and alternating-time properties for exogenous coordination models," Ph.D. dissertation, Technische Universität Dresden, Germany, 2012.

[43] C. Baier, T. Blechmann, J. Klein, S. Klüppelholz, and W. Leister, "Design and verification of systems with exogeneous coordination using Vereofy," in *Proc. 4th Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2010), Part II*, ser. Lecture Notes in Computer Science, vol. 6416. Springer-Verlag, 2010, pp. 97–111.

[44] P. C. Ölveczky and J. Meseguer, "Semantics and pragmatics of Real-Time Maude," *Higher-Order and Sym-*

bolic Computation, vol. 20, no. 1-2, pp. 161–196, 2007.

[45] P. C. Ölveczky and S. Thorvaldsen, "Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude," in *Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, ser. Lecture Notes in Computer Science, vol. 4468. Springer, 2007, pp. 122–140.

[46] E. B. Johnsen, O. Owe, J. Bjørk, and M. Kyas, "An object-oriented component model for heterogeneous nets," in *Proc. 6th International Symposium on Formal Methods for Components and Objects (FMCO'07)*, ser. Lecture Notes in Computer Science, vol. 5382. Springer, 2007, pp. 257–279.

[47] W. Leister, X. Liang, S. Klüppelholz, J. Klein, O. Owe, F. Kazemeyni, J. Bjørk, and B. M. Østvold, "Modelling of biomedical sensor networks using the Creol tools," Norsk Regnesentral, Oslo, Norway, Report 1022, 2009.

[48] J. Meseguer and G. Rosu, "The rewriting logic semantics project: A progress report," in *FCT*, ser. Lecture Notes in Computer Science, O. Owe, M. Steffen, and J. A. Telle, Eds., vol. 6914. Springer-Verlag, 2011, pp. 1–37.

[49] E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa, "Simulating concurrent behaviors with worst-case cost bounds," in *Proc. 17th International Symposium on Formal Methods (FM'11)*, ser. Lecture Notes in Computer Science, M. Butler and W. Schulte, Eds., vol. 6664. Springer, 2011, pp. 353–368.

[50] E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa, "A formal model of object mobility in resource-restricted deployment scenarios," in *Proc. 8th International Symposium on Formal Aspects of Component Software (FACS 2011)*, ser. Lecture Notes in Computer Science, F. Arbab and P. Ölveczky, Eds. Springer-Verlag, 2012, to appear.

[51] A. Stam *et al.*, "Project deliverable D6.4: Validation," EU project IST-33826 CREDO: Modeling and Analysis of Evolutionary Structures for Distributed Services, Tech. Rep., 2009, accessed June 18, 2012. [Online]. Available: http://projects.cwi.nl/credo/publications/Deliverables/DeliverableD6-4.pdf

[52] A. Lomuscio, H. Qu, and F. Raimondi, "MCMAS: A model checker for the verification of multi-agent systems," in *Proc. 21st International Conference on Computer Aided Verification, CAV'09*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 682–688.