Power is Overrated, Go for Friendliness! Expressiveness, Faithfulness and Usability in Modeling - The Actor Experience

Marjan Sirjani[1,2]

[1] School of Innovation, Design and Engineering, Mälardalen University, Sweden
[2] School of Computer Science, Reykjavik University - Iceland

Abstract. Expressive power of a language is generally defined as the breadth of ideas that can be represented and communicated in a language. For formal languages, the expressive power has been evaluated by checking its Turing completeness. In a modeling process, apart from the modeling language we have two other counterparts, the system being modeled and the modeler. I argue that faithfulness to the system being modeled, and usability for the modeler are at least as important as the expressive power of our modeling language, specially because most of the modelling languages used today are highly expressive. I call faithfulness and usability together friendliness. I show how we used the actor-based language Rebeca in modeling different applications, where it is friendly, and where it is not; and how the features of the language and its friendliness may help in analysis of the model, and synthesis of the system based on the model.

Foreword

People have different ways of thinking, what seems simple, clear and understandable to me may seem highly complicated and convoluted to others. When we tell a story in our words we make a different model of the same concept, and this new model may give a better insight to certain audience. That is why people talk about the same concept again and again in different ways. I see three counterparts involved when you tell a story, the audience, the way you tell the story, and the story itself. I got to learn that all the three can be equally important. Edward has a wealth of knowledge, and a wide range of expertise. One of his several qualities is the way he tells the stories, he says what I want to say in a much better way! This gives me more courage to write, even if others have already told my story, I may say it in a better way, at least for a certain audience.

1 Introduction

Why yet another modeling language? I've seen this question at so many occasions, especially asked by people in the formal methods community. The main reason that this question is asked is because the tradition in theoretical computer science is to compare two languages based on their expressive power. Expressive power is generally defined as the breadth of ideas that can be represented and communicated in a language. One way that has been used for evaluating the expressive power of a language is checking the Turing-completeness. Turing completeness was not enough and the community moved towards other ways of comparing expressiveness, mostly based on mutually encoding the formalisms into each other. But most of the modeling languages we work with are highly expressive, and may have equivalent expressive powers. So, why yet another modeling language?

Turing completeness and most of the other ways of comparing languages checks computability, and nowadays interaction; the focus here is on the machines world. I can see two other major counterparts in modeling, the system that is modeled and the modeler. A modeling language has to be evaluated by its faithfulness to the system it is modeling, and usability for the modeler. I call usability and faithfulness together friendliness, friendliness to the system and friendliness to the modeler. What theoretical computer scientists are missing is the friendliness of the languages.

Since the main complexity of the modeling job is the computation part, it is natural to focus on that part. Moreover people tend to focus on parts that they understand better and are more familiar with. When we are working with more and more complicated applications with heterogeneous components and different technologies, then I believe friendliness of our modeling languages will become at least as important as their expressive power.

We also have to remember that the goal of building a model is usually doing analysis and/or building or synthesizing the system based on the model. So, analyzability is crucial. Expressive power and friendliness both affect analyzability and synthesizability. Sometimes faithfulness criteria may guide us to a less expressive language, and that may help in improving the analyzability (similar to domain-specific languages). Moreover, friendliness can give us a good traceability, from the model to the system. If we find a problem in the model, then we can trace it back into the system more easily. So, apart from expressiveness, friendliness can be a criterion for choosing the modeling language we want to use.

I have to add that there are different communities that consider modeling in all its aspects. For example, modeling is an important part in Software Engineering. The object-oriented paradigm came with the winning slogan of *decreasing the semantic gap* between the real world and the program, i.e., faithfulness. If we focus on expressive power we would be still programming in assembly languages.

Faithfulness and Usability Faithfulness is about the similarity of the model and the system. In most places it is defined as the degree of detail incorporated in the model [1]. What I mean in this paper by faithfulness of a modeling language is whether and how the structures and features that are supported by the modeling language match the needs of the domain of the system being modeled, and how much this helps in having a more natural mapping between the model and the system. More precisely, we can define faithfulness based on the definition of model of computation. A collection of rules that govern the execution of the [concurrent] components and the communication between components is called a model of computation (MoC) [2]. We say a modeling language is faithful to a system if the model of computation supported by the language matches the model of computation of [the features of interest of] the system. Faithfulness can be seen as the key motivation behind domain-specific languages.

Sometimes I use the term "*model*" when you expect to see "*modeling language*". This is where I mean the model of computation. The structures, features, and flow of control provided and imposed by your modeling language can shape your model. As they say languages can shape your thoughts.

In synthesis, we make a model, prior to building the system itself, to help us building the system. In the model we incorporate all the properties of interest. So, faithfulness is defined as how much the system is faithful to the model. This is what is common in engineering domains. In analysis, if the system already exists, we make an (abstract) model of the system to help us perform different kinds of analysis. This is the type of modeling that scientists are more familiar with. We can look at faithfulness in both directions, faithfulness of the system towards the model, and the model towards the system.

In ISO 9241 [3], usability is defined as the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. Effectiveness is accuracy and completeness with which users achieve specified goals; and efficiency is resources expended in relation to the accuracy and completeness with which users achieve goals. Satisfaction is freedom from discomfort and positive attitudes towards the use of the product. In this paper, I do not discuss usability in an extensive manner. I can only explain my experience through years, as we have not yet run any scientific experiment for evaluation usability of different modeling languages.

Edward and Modeling. The first time that I have seen a truly convincing answer for me to "why yet another modeling language?", was a text by Edward in the Ptolemy book [2] "An important part of a science, quite complementary to the scientific method, is the construction of models. Models are abstractions of the physical reality, and the ability of a model to lend insight and predict behavior may form the centerpiece of a hypothesis that is to be validated (or invalidated) by experiment. The construction of models is itself more an engineering discipline than a science. It is not, fundamentally, the study of a system that preexists in nature; it is instead the human-driven construction of an artifact that did not previously exist. A model itself must be engineered. Good models can even reduce the need for measurement, and therefore reduce the dependence on the scientific method." The keywords for me were "engineering", a "human-driven construction" which brings in the modeler and shows its importance, and "the ability of a model to lend insight" which I think can depend on the faithfulness as much as the expressiveness of the modeling language.

Actors and Friendliness. In this paper, I will explain how the actor-based [4] language Rebeca [5–8] is used for modeling and analysis of different domains of applications, and where and how it has been more faithful and usable, and where it has multiple shortcomings. I will not cover a comparison between modeling each of the applications using Rebeca versus modeling the same application using other modeling languages. The interested reader can find the comparisons in corresponding papers published on each application. For each application domain I will explain the mapping between the entities and concepts in the real world, and the ones in the Rebeca model. The interesting and important properties that have to be verified or analyzed in each domain is not always trivial. For each application I will explain the property that is checked, and the analysis that is done.

In the next section there is a short description of Rebeca and Timed Rebeca. In Section 3, I will explain how we used Timed Rebeca in modeling Sensor networks and check the schedulability [9, 10]. In Section 4, I will describe how extensions of Rebeca is used for analyzing different network protocols [11, 12]. In Section 5, I view Network on Chip (NoC) as an example of track-based traffic systems and show how we used Timed Rebeca in evaluating different routing algorithms [13–15]. In Section 6, I will give a short overview of friendliness, analyzability and other features of Rebeca.

The contribution of this paper is not presenting a novel technique or a new model, it is telling an already told story in a different way. The message is where and how friendliness of a language can help in modeling and analysis, and the target audience are mainly those who are looking for a modeling language for analyzing their application.
Disclaimer: Most of the technical material in this paper is taken from published or draft papers, in some places the sentences are copied without using quotes.

2 The Actor-based Language, Rebeca

Rebeca (Reactive Object Language) [5, 7] is an actor-based language based on Hewitt and Agha's actors [16, 4]. Actors are units of concurrency, with no shared variables, communicating by asynchronous messages. There is no explicit receive statement, and send statements are non-blocking. Rebeca is an imperative language, with Java-like syntax. There is only one single thread of execution in each actor and one message queue. The actor takes a message from top of its message queue, and executes the corresponding method (called *message server*) non-preemptively. If you see messages as events, then Rebeca model can be seen as an event-driven model. The execution of message servers is also similar to *atomic asynchronous call-backs* in the context of Java Script.

In Timed Rebeca (the real-time extension of Rebeca) [17, 18, 8], instead of a message queue we have a message bag where messages are tagged with their time-stamps (sometimes I use message buffer as a more general term instead of message queue or bag). We consider synchronized local clocks throughout the model for all the actors (you can read it as a global time). The sender tags a message with its own local time, at the time of sending. This can be seen as *model time* in Ptolemy.

A Rebeca model consists of a number of *reactive classes*, each describing the type of a certain number of *actors* (called *rebecs*, we use both terms rebec and actor interchangeably in the Rebeca context). Each reactive class declares the size of its message buffer, a set of *state variables*, and the messages to which it can respond. The local state of each actor is defined by the values of its state variables and the contents of its message buffer. Each actor has a set of *known rebecs* to which it can send messages. Reactive classes have constructors, with the same name as their reactive class. They are responsible for initializing the actor's state variables and putting initially needed messages in the message buffer of that actor. See Figure 1 for an abstract syntax of Timed Rebeca.

$$
\begin{aligned}
Model &::= Class^{*}\ Main \\
Main &::= \textbf{main}\ \{\ InstanceDcl^{*}\ \} \\
InstanceDcl &::= className\ rebecName(\langle rebecName\rangle^{*}) : (\langle literal\rangle^{*}); \\
Class &::= \textbf{reactiveclass}\ className\ \{\ KnownRebecs\ Vars\ MsgSrv^{*}\ \} \\
KnownRebecs &::= \textbf{knownrebecs}\ \{\ VarDcl^{*}\ \} \\
Vars &::= \textbf{statevars}\ \{\ VarDcl^{*}\ \} \\
VarDcl &::= type\ \langle v\rangle^{+}; \\
MsgSrv &::= \textbf{msgsrv}\ methodName(\langle type\ v\rangle^{*})\ \{\ Stmt^{*}\ \} \\
Stmt &::= v = e;\ \mid v = ?(e, \langle e\rangle^{+});\ \mid Call;\ \mid \textbf{delay}(t);\ \mid if\ (e)\ \{\ Stmt^{*}\ \}[else\ \{\ Stmt^{*}\ \}] \\
Call &::= rebecName.methodName(\langle e\rangle^{*})\ [\textbf{after}(t)]\ [\textbf{deadline}(t)]
\end{aligned}
$$

Fig. 1. Abstract syntax of Timed Rebeca (from [19]). Angled brackets ⟨...⟩ are used as meta parenthesis, superscript + for repetition at least once, superscript $*$ for repetition zero or more times, whereas using ⟨...⟩ with repetition denotes a comma separated list. Brackets [...] indicates that the text within the brackets is optional. Identifiers *className*, *rebecName*, *methodName*, *v*, *literal*, and *type* denote class name, rebec name, method name, variable, literal, and type, respectively; and e denotes an (arithmetic, boolean or nondetermistic choice) expression.

The way an actor responds to a message is specified in a *message server*. The state of an actor can change during the executing of its message servers through assignment statements. An actor makes decisions through conditional statements, communicates with other actors by sending messages, and performs periodic behavior by sending messages to itself. Since communication is asynchronous, each actor has a *message buffer* from which it takes the next incoming message. An actor takes the first message from its message buffer, executes its corresponding message server in an isolated environment, takes the next message (or waits for the next message to arrive) and so on. A message server may have a *nondeterministic assignment* statement which is used to model the nondeterminism in the behavior of a message server. Finally, the `main` block is used to instantiate the actors of the model. Note that Rebeca does not support dynamic actor creation, and all the actors of a model must be defined in the main block.

Timed Rebeca adds three primitives to Rebeca to address timing issues: *delay*, *deadline* and *after*. A *delay* statement models the passage of time for an actor during execution of a message server. Note that all other statements of Timed Rebeca are assumed to execute instantaneously. The keywords *after* and *deadline* are used in conjunction with a method call. The term `after(n)` indicates that it takes n units of time for a message to be delivered to its receiver. The term `deadline(n)` expresses that if the message is not taken in n units of time, it will be purged from the receiver's message bag automatically.

Actors in Ptolemy and Rebeca. Actors in Ptolemy are more like components in a Software Engineering terminology. In Ptolemy, actors have ports, they read and write to and from their ports, while in Rebeca actors send messages to each other knowing each others names (like objects in object-oriented languages). Ptolemy actors may have more than one port, while in Rebeca there is only one message buffer.

Note that in Ptolemy you have *directors* that coordinate the behavior of actors. Through that coordination you are able to impose an order on the execution of actors and make the model deterministic. You can also make different models of computation. Rebeca and Timed Rebeca can be seen as specific models of computation in Ptolemy.

Rebeca is initially designed for analysis, and hence supports features for making a model of an existing system. The language allows non-deterministic assignments, and the model checking tools consider non-deterministic order of execution (or an interleaved model of concurrency). Ptolemy is initially designed for synthesis, and hence there are powerful techniques to avoid non-determinism. When synthesizing, you desire, and you do your best to make your model functions deterministically, no matter how the environment (and the underlying technology on which your system will be built on) is non-deterministic. Despite of all, both languages can be used in different ways, you are able to make a deterministic model in Rebeca, and a non-deterministic one in Ptolemy. Rebeca models can be and are used for synthesizing (after analyzing your abstract designs), and Ptolemy models are analyzed (before synthesizing your system).

3 Wireless Sensor Network Applications and Schedulability

Wireless sensor and actuator networks (WSANs) are built from a collection of nodes that collect the data from the surroundings to achieve specific application objectives. A WSAN application is a distributed system where multiple nodes are used to monitor the surroundings like temperature, humidity, pressure, or position, and perform various tasks like smart detecting, and target tracking.

WSANs can provide low-cost continuous monitoring. However, building WSAN applications is particularly challenging because of the complexity of concurrent and distributed programming, networking, real-time requirements, and power constraints. It can be hard to find a configuration that satisfies these constraints while optimizing resource use [9]. WSAN applications are sensitive to timing, with soft deadlines at each step of the process that are required to ensure correct and efficient operation.

Several software platforms have been developed specifically for WSANs [20]. Among these, the most accepted platform is the TinyOS [21], which is an open-source operating system designed for wireless embedded sensor networks. TinyOs is based on an event-driven execution model that enables fine-grained power management strategies.

A sensor node, is a node in a wireless sensor network that is capable of performing some processing, gathering sensory information and communicating with other connected nodes in the network. Each sensor node consists of independent concurrent entities, including CPU, sensor, and radio system. These sensor nodes are connected via a wireless communication device which uses a transmission control protocol. Interactions between entities, both within a node and across nodes, are concurrent and asynchronous.

Modeling Sensor Nodes and Communication Medium in Rebeca[3]. We consider sensor nodes in WSAN applications, and we also model the network between these nodes. A sensor node is responsible for monitoring, it collects data, perform necessary processing, and then send the data to another node via network. A sender node has concurrent components performing the *sensing*, *data processing* and *transmitting* the data. In addition to processing the data provided by the sensor component, there are also *miscellaneous* tasks that the processing unit in a node has to handle. So, we have four actors (concurrent and asynchronously executing objects) which all are located in a sensor node (see Figure 2 for a visual mapping of real world entities and actors in the Rebeca model, and see Figure 3 for the Rebeca code):

– Sensor actor for sensing,
– CPU actor for processing,
– Communication Device actor for transmitting (CD), and
– Misc actor for performing miscellaneous tasks.

In some applications a sensor node works as a router and passes the data that it has received, this is done by the Communication Device.

We have a fifth actor named Wireless Medium that models the communication medium. Wireless Medium informs the Communication Device of the status

[3] In some places I say Rebeca when I mean Timed Rebeca

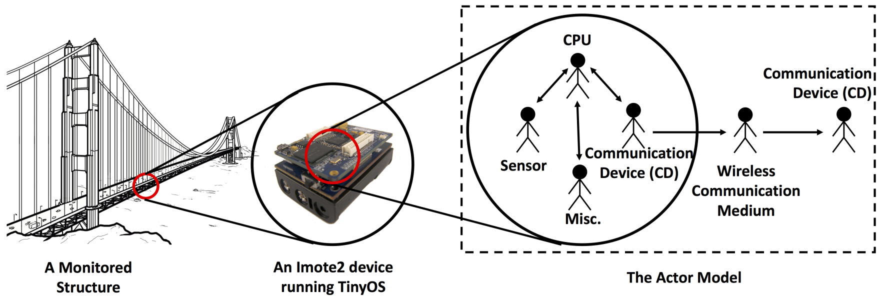

CPU
Sensor
Communication Device (CD)
Misc.
Wireless Communication Medium
Communication Device (CD)
A Monitored Structure
An Imote2 device running TinyOS
The Actor Model

Fig. 2. Modeling the behavior of a WSAN application in its real-world installation in the actor model (from [9])

of the network and performs broadcasting of the data. Each of these two tasks are modeled as a message server (i.e. event handler) in Rebeca. The details of the communication protocol, like the implementation of the Media Access Control (MAC) level, is modeled in the Communication Device actor. Different protocols that are modeled in the Communication Device actor basically trigger the two events of the Wireless Medium (asking for the status of the network, and asking for broadcasting the data) in different ways. As a result, different implementations of communication protocols can be replaced without significantly impacting the remainder of the model. During the application design phase, different components, services, and protocols may be considered. For example, TDMA [23] as a MAC-level communication protocol may be replaced by B-MAC [24] with minimal changes.

Timed Rebeca code. Figure 3 shows an abstract version of the Timed Rebeca code of the WSAN application. The main activity of this model is started by executing `sensorLoop` (line 16) of the `Sensor` actor (line 10). In this loop, based on the specified sampling rate, data is acquired by `Sensor` and it is sent to `CPU` (line 18). There is the same behavior in `Misc` (line 21). These two actors send messages to `CPU` (line 22). The actor `CPU` handles the messages received from `Sensor` and `Misc` by the `sensorEvent` and `miscEvent` message servers respectively (lines 28 and 40). The message server `sensorEvent` starts the processing of the acquired data by sending a `sensorTask` message (line 29). In `sensorTask` (line 31), the schedulability of processing of acquired data is checked, it is packed into one packet, and the packed data is sent by the communication device of this node if it reaches the limit which is specified by `bufferSize` (lines 36-37).

The communication protocol between nodes is implemented in the actor `CommunicationDevice` (line 41). The Rebeca model for TDMA and B-MAC communication protocols can be found in [9]. In the current implementation shown in Figure 3, before sending data, the freedom of the communication device is checked, then the needed messages are scheduled for sending data.

The effect of the wireless communication and transmission conflict is modeled by the actor `WirelessMedium` (line 62). Communication devices send `broadcast`

```
env int samplingRate = 25;
env int numberOfNodes = 6;
env int bufferSize = 2;
env int sensorTaskDelay = 2;
env int OnePacketTransmissionTime = 7;
env int miscTaskDelay = 10;
env int tmdaSlotSize = 10;
env int miscPeriod = 120;
env int packetMaximumSize = 112;
reactiveclass Sensor(10) {
   knownrebecs { CPU cpu; }
   Sensor() { self.sensorFirst(); }
   msgsrv sensorFirst() {
      self.sensorLoop() after(?(10,
          20, 30));
   }
   msgsrv sensorLoop() {
      int period = 1000 /
          samplingRate;
      cpu.sensorEvent(period);
      self.sensorLoop()
          after(period);
 }  }
reactiveclass Misc(10) { ... }
reactiveclass CPU(10) {
   knownrebecs {
      CommunicationDevice senderDev,
          receiverDev;
      Sensor sensor;}
   statevars { int
       collectedSamplesCounter; }
   CPU() { collectedSamplesCounter =
       0; }
   msgsrv sensorEvent(int period) {
      self.sensorTask(period,
          currentMessageWaitingTime);
   }
   msgsrv sensorTask(int period, int
       lag) {
      int tmp = period - lag -
          currentMessageWaitingTime;
      assertion(tmp >= 0);
      delay(sensorTaskDelay);
      collectedSamplesCounter += 1;
      if (collectedSamplesCounter ==
          bufferSize){
         senderDev.send(receiverDev,
             0, 1);
         collectedSamplesCounter =
             0;
   }   }
  msgsrv miscEvent() {
      delay(miscTaskDelay); }
reactiveclass CommunicationDevice
    (10) {
   knownrebecs { WirelessMedium
       medium; }
   statevars {
      byte id;
      int sendingData;
      int sendingPacketsNumber;
      CommunicationDevice
          receiverDev;}
   CommunicationDevice(byte myId) {
      id = myId;
      sendingData = 0;
      sendingPacketsNumber = 0;
      receiverDev = null;}
   msgsrv send(CommunicationDevice
       receiver, int data, int
       packetsNumber) {
      assertion(receiverDev == null);
      sendingPacketsNumber =
          packetsNumber;
      receiverDev = receiver;
      sendingData = data;
      medium.getStatus();}
   msgsrv receiveStatus(boolean
       result) { ... }
   msgsrv receiveResult(boolean
       result) { ... }
   msgsrv
       receiveData(CommunicationDevice
       receiver, int data, int
       receivingPacketsNumber) { ...
       }
reactiveclass WirelessMedium(5) {
   statevars {
      CommunicationDevice senderDev;
      CommunicationDevice
          receiverDev;
      int maxTraffic;}
   WirelessMedium() {
      senderDev = null;
      receiverDev = null;
      maxTraffic = (125 * 1024) / 8;
   }
   msgsrv getStatus() { ... }
   msgsrv
       broadcast(CommunicationDevice
       receiver, int data, int
       packetsNumber){ ... }
   msgsrv broadcastingIsCompleted() {
      senderDev = null;
      receiverDev = null;
}   }
main {
 WirelessMedium medium():();
   CPU cpu (sensorNodeSenderDevice,
       receiver, sensor):();
   Sensor sensor(cpu):();
   Misc misc(cpu):();
   CommunicationDevice
       sensorNodeSenderDevice(medium):((byte)1);
   CommunicationDevice
       receiver(medium):((byte)0);}
```

Fig. 3. The Rebeca model of a WSAN application (based on the code in [22])

messages (line 73) to the wireless medium to send data to other communication devices and the receivers of broadcast data send `broadcastingIsCompleted` (line 74) to inform receiving of the data successfully.

Faithfulness. In the WSAN example, all the counter-parts that are running concurrently in the system are modeled as actors: sensor, CPU, communication device, Misc, and wireless medium. The focus is on the schedulability of tasks. Each actor asks the CPU for execution of some tasks and the question is if the CPU can handle all the tasks without missing any deadlines. So, what has been modeled accurately are different services that are requested from CPU, and their timing. We also had to model the communication medium as an actor because the status of the network affects the overall behavior. TinyOS and Rebeca match perfectly in their MoC. There are no wait or receive statements, event-handlers are executed non-preemptively, and there are no priority queues.

Usability. As for usability of Rebeca in modeling WSAN applications, we can claim effectiveness, efficiency and satisfaction. Users can achieve their goal of schedulability analysis in a complete and accurate way (effectiveness). The model can capture all the necessary details, and the model checking tool provides necessary information more accurately than alternative techniques of simulation or mathematical analysis. Efficiency relates to the time that modeler needs to achieve her goals. For a software engineer or a computer scientist, writing Java- or C-like codes is simpler and takes less time comparing to writing mathematical formulas. Also, comparing to simulation tools, by using Rebeca we build more abstract models, and hence we spend less time. Based on our experience, the majority of software engineers and computer scientists prefer program-like syntax, and hence Rebeca brings in positive attitude and satisfaction. Moreover, Faithfulness brings in usability, a natural, and in most cases one-to-one mapping of the constructs in WSAN applications into the Rebeca model makes the process effective, efficient, and with least hassle.

Reusability, and Modeling Different Protocols. For modeling different protocols, we only need to change the code of actor `Communication Device`. By using Rebeca, we preserve the modular design of the protocol, so, we improve reusability. When we use other paradigms for modeling network protocols, like process algebra or automata, we usually need to spread out the functionality of one module of the system throughout different modules of the model. This will jeopardize reusability.

TDMA protocol defines a cycle, over which each node in the network has one or more chances to transmit a packet or a series of packets. If a node has data available to transmit during its allotted time-slot, it may be sent immediately. Otherwise, packet sending is delayed until its next transmission slot.

The periodic behavior of TDMA slot is handled by a message server which sets and unsets a flag to show that whether the node is in its allotted time-slot or not. Upon entering into it's slot, a device checks for pending data to send and schedules a message to be sent at the end of the time-slot. On the other hand, when `CPU` sends a packet (message) to a `Communication Device`, the message is added to the other pending packets which are waiting for the next allotted time slot.

In contrast to TDMA, in B-MAC, RCD tries to detect free channel status and send data upon receiving a request from CPU. In the case of detecting free channel, the data is sent immediately. This way, collision may occurs; so, `Communication Device` has to wait for some amount of time and resend data. B-MAC protocol does not need complicated and expensive synchronization methods. It also avoids data fragmentation. So, it would be more complicated to coordinate long messages and B-MAC expects short messages, which is common for information of WSAN nodes.

Schedulability Analysis. In the application we require that all the periodic tasks (sample acquisition, data processing, and radio packet transmission) are completed before the next iteration starts. So, this defines the deadline for each task. The goal is to have a higher sampling rate or a larger number of nodes without violating schedulability constraints.

The configuration of this model is specified by the values of the environment variables (lines 1 to 7 in Figure 3). Based on these values, there are six nodes in the environment (line 2) and the sampling rate of the nodes is 25 samples per 1000 units of time (line 1). Each node packs two acquired data elements in one packet (line 3). The time spent for the internal activities of a node is specified in lines 4 to 6.

The Afra model checking tool verifies whether the schedulability properties hold in all reachable states of the system. If there are any deadline violations, a counterexample will be produced. A counterexample shows the sequence of states from an initial configuration that results in the violation. This information can be used to change the system parameters in order to avoid such situations, for example by increasing the TDMA time slot length or reducing the sampling rate.

TCTL model checking can be used to check the utilization of resources, for example we can check the utilization of the communication medium.

Scalability Challenges. One way of modeling WSAN using actor model is to instantiate actors for each node in the network. That may cause state explosion when doing the model checking. So, a main challenge is to find an effective and correct abstraction technique. In TDMA, the packet transmission of one sensor node does not interfere with the other sensor nodes. Having more sensor nodes only results in having shorter time slots, so the presence of sensor nodes can be abstracted and modeled by making time slots shorter. Using this abstraction, we only have to model one node which is in communication with the central node. So, verification of WSAN applications against schedulability and deadlock-freedom properties become feasible for networks in any size [10].

In B-MAC, the presence of sensor nodes can be abstracted and modeled as the possible number of collisions before a data communication is performed successfully [10]. Using this abstraction, only one sensor node which is in communication with the central node has to be considered for networks in any size. Any data transmission of this sensor node may encounter a collision. The maximum number of the collisions is the number of sensor nodes in the model. So, in the Rebeca code for `Communication Device`, for each data transmission we have a non-deterministic choice between a successful transmission or a collision. During model checking, in the case of collision, data transmission with zero, one, ..., up to n collisions are considered where n is the number of sensor nodes.

4 Mobile Ad-hoc Network Protocols and Finding Possible Faults

A Mobile Ad-hoc Network (MANET) is a wireless network consisting of mobile routers (and associated hosts) connected by wireless links, the union of which forms an arbitrary topology. The routers are free to move randomly and organize themselves arbitrarily, so, the network's wireless topology may change rapidly and unpredictably.

MANETs have different applications from military to managing disastrous situations where there is no network infrastructure and nodes can freely change their locations. Mobility is the main feature of MANETs which makes them powerful and at the same time error prone in practice. The process of protocol design is not straightforward. Since there is no base station or fixed network infrastructure, every node acts as a router and keeps the track of the previously seen packets to efficiently forward the received messages to desired destinations. In essence, MANETs need routing protocols in order to provide a way of communication between two indirectly-connected nodes. In the protocol, there has to be an algorithm for each node to continuously maintain the information required to properly route traffic.

MANETs are wireless sensor networks; but the differences between WSANs, discussed in Section 3, and MANETs are that in WSANs there is usually one sink (or base station) which collects the data, and there are fixed routes in the network (except when we have failures of nodes). In MANETs, nodes are continuously moving in any direction, and there is no fixed route between two nodes.

Routing protocols for MANETs are devised in a completely distributed manner and adaptive to topology changes, so, building reliable and efficient routing protocols is complicated and also crucial. The Ad-hoc On Demand Distance Vector (AODV) protocol is one of the most prominent routing protocol in MANETs. The AODV protocol has been evolved as new failure scenarios were experienced or errors were found in the protocol design.

Modeling MANETs in Rebeca. One of the challenges in modeling MANETs is representing the connectivity of pairs of nodes in the network. Two nodes are connected if they are within the wireless communication range. As the nodes are moving the network topology is changing all the time. Rebeca is extended in [12] to wRebeca, to address local broadcast and dynamically changing topology. In order to abstract the data link layer services, the wireless communications in the framework, namely local broadcast, multicast, and unicast, are considered to be reliable. So, a node can broadcast/multicast/unicast a message successfully to the nodes within its communication range, and the message delivery is guaranteed for the connected nodes to the sender. In the case of unicast, if the sender is located in the receiver communication range, it will be notified, otherwise it assumes that the transmission was unsuccessful so it can react appropriately.

Each node in the network is modeled as an actor, and the routing protocol is represented through the message servers of the actor. The network topology and its mobility are captured while analyzing the model, and are not explicitly modeled in the Rebeca code.

Rebeca Code. The wRebeca model of an abstract version of AODV is given in Figure 4. There is one reactive class, *Node*, representing the nodes in the network. In this protocol, routes are built upon route discovery requests and maintained in nodes routing tables for further use. In message server *rec-newpkt* (line 14), whenever a node intends to send a data packet, it looks up in its routing table to see if it has a valid route to the intended destination. In case it finds a route, it sends the data packet through the next-hop specified in that route (line 16-17), otherwise it starts a route discovery by broadcasting a route request, *rec-rreq*, after increasing its sequence number (line 18-21).

In message server *rec-rreq* (line 23), whenever a node receives a new routing packet, it updates its routing table with new information to keep it up-to-date. The forward messages contain the route back to the *source*, while the backward messages carry the route information towards a *destination*. While the forward packet proceeds towards the destination, a *backward path*, a path to *source* from *destination*, is constructed. In message server *rec-rreq*, every node upon receiving a packet looks up its routing table and if it has a route to the requested destination it would reply by sending a *rec-rrep* message (line 31). Otherwise, it continues route discovery by re-sending the *rec-rreq* message, after increasing the hop-count. There is an upper limit for the hop-count, after that the algorithm gives up on that route. The *unicast* message (line 31) will be delivered successfully (*succ* in line 32) if the receiver node is in the access range, or the delivery can fail (*unsucc* in line 36) if the receiver node is not in the access range.

In message server *rec-rrep* (line 48), whenever a node receives a message it updates its routing table accordingly to construct the *backward path*. When it reaches the source, a bidirectional route has been formed and the data packet can be sent towards the destination through the next-hops in the routing tables. In addition to the above message servers, there is message server *rec-rerr* (line 70) that is called whenever a node fails to send a packet through a *valid* route, in order to inform other interested nodes in the broken route about the failure. Due to the mobility of the nodes this may happen often.

Faithfulness. For MANETs, we modeled the network nodes as actors. Nodes send asynchronous messages to each other, and the protocol is modeled by message servers. The MoCs match perfectly, except that Rebeca in its core form, does not support broadcast or multicast. But broadcast and multicast are both asynchronous, and non-blocking from the sender side; and we do not need any explicit receive statement in the receiver side. So, the crucial rules of the MoC stay unchanged, i.e., in the semantics the main transition rule which is taking

```
reactiveclass Node()
{
  statevars
  {
    int sn, ip;
    int[] dip, dsn, route_state,
         hops, nhops,
  }
  Node(int i, boolean starter)
  {
    /* initializing the route table
         variables*/
    if(starter==true) {
      unicast(self,rec-newpkt(7,2));
  } }
  msgsrv rec-newpkt(int data ,int
       dip_)
  {
    if(route_state[dip_]==1) {
      /* valid route to dip forward
           packet */
    } else {
      /* no valid route to dip send a
           new rout discovery request
           */
      sn++;
      rec-rreq(0, dip_, dsn[dip_],
           self, sn, self, 5);
  } }
  msgsrv rec-rreq (int hops_, int
       dip_ , int dsn_ , int oip_ ,
       int osn_ , int sip_, int
       maxHop)
  {
    boolean gen_msg = false;
    /* evaluate and update the routing
         table, decide whether a new
         rreq should be generated */
    if (gen_msg == true) {
      if (ip == dip_) {
        sn = sn+1;
        /* unicast the RREP towards
             oip of the RREQ */
        unicast(nhop[oip_],rec-rrep(0
             , dip_ , sn , oip_ ,
             self))
        succ:
        {
          route_state[oip_] = 1; break;
        }
        unsucc:
        {
          if(route_state[oip_] == 1) {
            /* error recovery procedure
                 */
          }
          route_state[oip_] = 2;
        }
      } else {
        hops_ = hops_ + 1;
        if(hops_<maxHop) {
          rec-rreq(hops_, dip_, dsn_,
               oip_, osn_, self,
               maxHop);
  } } }}
  msgsrv rec-rrep(int hops_ ,int dip_
       ,int dsn_ ,int oip_ ,int sip_)
  {
    boolean gen_msg = false;
    /* evaluate and update the routing
         table, decide whether a new
         rreq should be generated */
    if(gen_msg == true)
    {  if(ip == oip_ )
      { /* this node is the originator
           of the corresponding RREQ,
           a data packet may now be
           sent */ }
      else {
        hops_= hops_+1;
        unicast(nhop[oip_],
             rec-rrep(hops_, dip_,
             dsn_, oip_, self))
        succ:
        {
          route_state[oip_]=1;
          break;
        }
        unsucc:
        {
          if(route_state[oip_] == 1) {
            /* error recovery procedure
                 */
          }
          route_state[oip_] = 2;
  } } } }
  msgsrv rec-rerr(int source_ ,int
       sip_, int[] rip_rsn)
  {
  /* regenerate rrer for invalidated
       routes */
} }
main
{
  Node node0(node1,node3):(0,true);
  Node node1(node0,node3):(1,false);
  Node node2(node3):(2,false);
  Node
       node3(node2,node0,node1):(3,false);
  constraints
  { and(con(node0,node1),
       con(node2,node3)) }
}
```

Fig. 4. The AODV specification given in wRebeca (based on the code in [25])

a message and executing the message server is not changed. Moreover, mobility of the nodes is captured at the level of state transition system at the time of analysis. This keeps the model simple. Different properties of the protocols can be checked using the model checking tool.

Usability. Usability of Rebeca in modeling network protocols depends on the goal. The modeling process can be performed efficiently and with satisfaction, each node is running concurrently and generally there are asynchronous communication. So, each node can be mapped to an actor. Communication protocols are usually written as algorithms or pseudo-codes in an imperative form, and can be naturally mapped to message servers in Rebeca. The effectiveness of the modeling depends on the goal: what kind of analysis has to be done, and what properties must be checked. Based on the properties we need to check we have to model different features of the system. We need reduction techniques to tackle state space explosion in the analysis phase. Comparing to alternative modeling paradigms, faithfulness of the model brings in usability.

Reusability and Modeling Different Network Protocols. Different versions of the AODV are modeled in wRebeca. For each version, the parts of the message servers related to updating the routing table is revised. The local data in the routing table must be adjusted based on the information that should be maintained for each version. Most of the code can remain unchanged.

Analyzing Wireless Ad-hoc Networks protocols. The goal in [12] is to find the conceptual mistakes in the protocol design, rather than the problems caused by an unreliable communication. A customized model checking tool is developed [26], and the loop-freedom property is checked while generating the state space. The reason for violating the property was maintaining multiple unconfirmed next hops for a route without checking them to be loop-free. Furthermore, the monotonic increase of sequence numbers and packet delivery properties are checked via model checking. The wRebeca team found a loop creation scenario in AODVv2 protocol (version 11) in 2016, and reported it to the AODV group. The AODV group confirmed the possibility of loop creation and released a new revised version of the protocol, and the authors are acknowledged [4]. Since then the new versions of the protocol are verified using wRebeca.

Scalability Challenges. While building the state space for analyzing a MANET protocol a few abstraction techniques are used. The first technique considers the network with a fix topology, ignoring the mobility of nodes. Then the actors that have the same neighbors and local states are considered identical. This way many states can be merged as the actors are no more distinguished by their identifiers. It is shown in [12] that the reduced transition system is strongly bisimilar to the original one, and the state space reduction is considerable. This technique is beneficial for finding an error during the design of a new version of a protocol. If we know that a certain topology leads to malfunctioning of a previous version

[4] The acknowledgment is at https://tools.ietf.org/html/draft-ietf-manet-aodvv2-16

of the protocol, we can check the new version of the protocol using that certain topology.

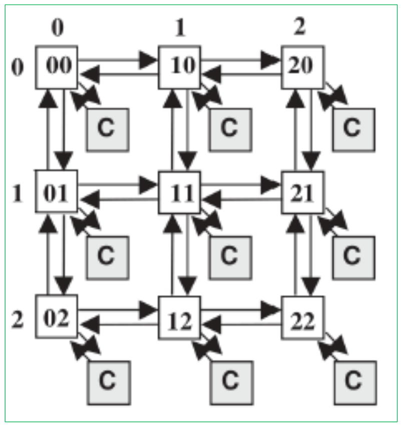
The above technique ignores the mobility of nodes and will not work if we have a dynamic topology. As an example of an effective design decision, in [12], changes in the topology are not captured at the level of wRebeca model. Instead, for analyzing the protocols, arbitrary changes in the underlying topology is considered while generating the state space. These random changes make the state space grow exponentially. To tackle the state space explosion, the states which are only different in their topologies are combined, and the topology-sensitive behaviors are captured by adding appropriate labels on the transitions. It is proved in [12] that the reduced transition system is branching bisimilar to the original one, and consequently a set of properties such as ACTL-X are preserved. Another way used to restrict the random changes in the topology, is allowing the modeler to specify constraints over the topology in the model.

5 Network on Chips and Routing

System-on-chip (SoC) designs provide integrated solutions to challenging design problems in the telecommunications, multimedia, and home electronics domains [27]. An SoC can be viewed as a micronetwork of components. The network is the abstraction of the communication among components and must satisfy quality-of-service requirements - such as reliability, performance, and energy bounds. Network on Chip (NoC) (an SoC paradigm) is a network of computational, storage and I/O resources, interconnected by a network of switches. Computing resources communicate with each other using addressed data packets routed to their destination [28]. In NoC designs, functional verification and performance evaluation in the early stages of the design process are suggested as ways to reduce the fabrication cost.

Modeling NoC in Rebeca. As an example of a NoC, we modeled and analyzed ASPIN (Asynchronous Scalable Packet switching Integrated Network), which is a fully asynchronous two-dimensional NoC design [29]. In an ASPIN design, each core is placed in a two-dimensional mesh and has (at most) four adjacent cores and four internal buffers for storing the incoming packets (one for each direction). Figure 5 shows the 2D mesh consisting of nine clusters (on the left), and a zoom-in picture of each cluster (on the right). The four (pairs of input and output) internal buffers are shown in the Figure.

Different routing algorithms have been proposed for the two-dimensional NoC design. Here, we consider the XY routing algorithm. Using the XY routing algorithm, packets are moving along the X direction first, and then along the Y direction, to reach their destination cores. In ASPIN, packets are transferred through channels, using a four-phase handshake communication protocol. The protocol uses two signals, namely *Req* and *Ack*, to implement this four-phase handshaking protocol. This way, to transfer a packet, first the sender sends a request by raising the *Req* signal along with the data, and waits for an acknowledgment which is the raising of the *Ack* signal by the receiver. In the third



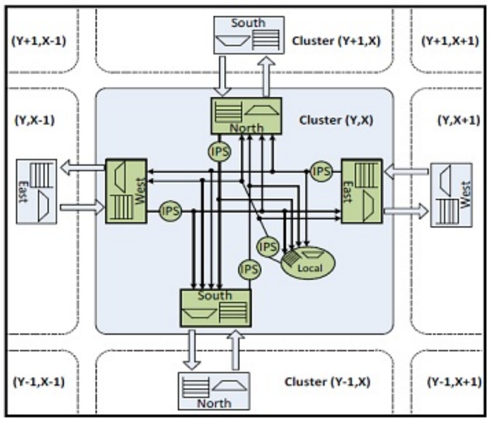


Fig. 5. A 2D mesh NoC (on the left), and a router in ASPIN [29] (on the right)

phase, when the sender gets the *Ack* from the receiver it will lower down the *Req* signal. Finally, in the fourth phase. when the receiver notices that the *Req* signal is lowered down it will lower down the *Ack* signal. So, after a successful communication all of the signals return to zero.

Timed Rebeca Code. The simplified version of the Timed Rebeca model of ASPIN is shown in Figure 6, which contains two different reactive classes: `Manager` and `Router`. The `Manager` (line 11) does not exist in real NoC systems, it is used here to model different scenarios of packet generation. In Figure 6, in function *testScenario*, two packets are generated, each contains two flits (lines 13-14 and 15-16). One packet is sent from the r00 router to r11 at the time 184 (the first flit), and 274 (the second flit), and the other packet is sent from r01 router to r11 at the time 18 (the first flit), and 110 (the second flit).

Each `Router` has four known rebecs which are its four neighbors (line 21). Its state variables include a composite id which is its X-Y position, buffer variables which show that the buffers are enabled or full, and a counter for the number of received packets (lines 24-25). Packets move through channels according to the four phase handshake communication protocol. Trying for the delivery of a packet started by sending an `inReq` message to a router. The receiver router accepts the packet if its input buffer is free. Upon accepting a packet, an acknowledgment is sent to its sender and an internal message is scheduled to process this packet. The time needed to do some of the processing or routing is modeled using *delay* or *after* constructs in the code. Processing of a packet takes place in message server `process` (line 35). If there is a packet for processing, based on

```
env byte inBufSize = 2;
env byte writeT = 2;
env byte readT = 6;
env byte flitNum = 2;
...
reactiveclass Manager(60){
  knownrebecs{
    Router r00, r10, r01, r11;
  }
  statevars{}
  Manager(){testScenario(); }
  void testScenario(){
    r00.inReq(4,1,1,1) after (184);
    r00.inReq(4,1,1,1) after (274);
    r01.inReq(4,1,1,2) after (18);
    r01.inReq(4,1,1,2) after (110);
  }
}
reactiveclass Router(60) {
  knownrebecs {
    Router N, E, S, W;
  }
  statevars {
    byte Xid, Yid, received;
        boolean[5] inBufFull,
             outBufFull;
        }
//---Comunication---
  msgsrv inReq (byte inPort, byte
       Xtarget, byte Ytarget, byte
       id){
  if (inBufFull[inPort] == false ){
    sendInAck((byte)(inPort + 2)%4,
         inAD);
    self.process(inPort, Xtarget,
         Ytarget,id, false,
         false)after((writeD *
         inBufSizeTest)+ readD);
    ...
  } else { ... }
}
msgsrv process(byte inPort, byte
     Xtarget, byte Ytarget,byte id,
     boolean isPushed, boolean
     justPush) { ...}
         ...
//---Routing Algorithm---
  byte XYrouting(byte Xtarget, byte
       Ytarget){
  if (Xtarget > Xid) //East
  else if (Xtarget < Xid) //west
  else if (Ytarget > Yid) //South
  else if (Ytarget < Yid) //North
  else outPort = 4; //the local
       buffer, arrived at destination
  return outPort;
}
//---Scheduling Algorithm---
  byte RRSched(byte outPort){
  byte[5] priorities = {4, 3, 1, 0,
       2};
  //turn = Number of the last input
       port which was its turn
  //passedFlit = Number of passed
       flits which was sent from
       "turn" to outPort
  if(BufFull[outPort]) return;
  if (passedFlit == 0){  // this
       flit is the header
  for(byte i=0 ; i<5 ; i++){
    //turn= according to priorities,
         choose next input port
         which is waiting for outPort
    outReqEnable[turn] = false;
    //Save turn for outPort
    passedFlit ++;
    if(passedFlit == flitNum){
      passedFlit = 0;
    }
    //save passedFlit for outPort
  }
  }else{// body of the packet
    outReqEnable[turn] = false;
    passedFlit ++;
    if(passedFlit == flitNum){
      passedFlit = 0;
    }
    //save passedFlit for outPort
  }
}
  -------------------------------------
  //Other auxiliary Functions &
       Message Servers
}
main {
  Manager m(r00,r01,r10,r11):();
  Router
       r00(m,r01,r10,r01,r10):(0,0);
  Router
       r10(m,r11,r00,r11,r00):(1,0);
  Router
       r01(m,r00,r11,r00,r11):(0,1);
  Router
       r11(m,r10,r01,r10,r01):(1,1);
}
```

Fig. 6. The Rebeca model of an ASPIN NoC (based on the code in [14])

the routing algorithm one of the *outPorts* is selected to send the packet to the appropriate neighbor. Routing is based on the XY algorithm, and the output port for routing a packet is computed by the function `XYrouting` (line 38). The scheduling algorithm is captured in the function `RRSched` (line 47). The 2D mesh of this model is formed in the `main` block of the model by setting known rebecs based on the locations of the routers (lines 77-80).

Faithfulness. ASPIN is a GALS NoC design, with synchronous behavior within each node and asynchronous message passing between nodes. So, we model each node (router and the core) as an actor, and the MoCs of ASPIN and Rebeca match, and a faithful model of NoC can be built. One can observe that within a router different ports can be running concurrently, but we did not model each port as an actor to avoid state space explosion. Reading from each input port and putting the packet into the correct output port is done using a round-robin scheduling policy which is modeled in the code. We do not lose any interesting property with this abstraction.

Usability. In a high level of abstraction NoC can be mapped to Rebeca efficiently. We showed that despite the high level of abstraction the results are consistent with hardware simulation results in the literature, so, the approach is effective. In the NoC project, extended versions of the model including the communication protocol and more detailed versions of the scheduling algorithm are developed in later phases. Adding more details (like buffer length, packet length and flit number, packet generation delay, more precise communication protocol) results in more precise measurements, showing effectiveness. Naturally, debugging the Rebeca code becomes more difficult when the code becomes more detailed. On the positive side, more details can be added to the model in an iterative and incremental way which is not a capability supported by all available hardware simulation tools. Our analysis technique is based on model-checking, it captures the simultaneity of the events (which is modeled using interleaving), while hardware simulation tools are not capable of that. As for satisfaction, a hardware designer may not be comfortable with programming in a C- or Java-like language.

Reusability and Modeling Different Routing Algorithms. Modeling different routing algorithms in Rebeca can be done efficiently; we have to change the routing function in the code. The rest of the code can be reused. Routing algorithms can be classified into deterministic and adaptive routings. In a deterministic routing there can only be one path between a source and a destination, whereas in an adaptive routing more than one possible path may exist and the algorithm considers the conditions of the dynamic network to decide in which direction a packet should be transferred. The XY algorithm is a deterministic algorithm, Odd-Even routing is an adaptive one, and DyAD routing chooses dynamically between a deterministic or an adaptive algorithm, based on the different network congestion conditions.

Odd-Even routing algorithm is based on Odd-Even turn model [30]. According to Odd-Even turn model north-to-west and south-to-west turns are prohibited in routers located in an odd column and east-to-south and east-to-north turns are prohibited in routers located in an even column. The restrictions are enforced to ensure deadlock freedom. For routing a packet, each router decides between two legitimate downstream neighbors based on the number of the empty slots in their input buffer. The neighbor with more empty slots will be selected. In this algorithm, each router keeps track of the number of packets in input buffer of each of its neighbors. In the Rebeca model [15], whenever the size of an input buffer of a router changes, it informs its corresponding upstream neighbor by sending a message.

In DyAD routing, each router monitors the occupation ratio of its input buffers (except for the local buffer). Whenever one of the buffers reaches a predefined congestion threshold a mode flag is set to inform the corresponding neighboring about the congestion. On the other hand, each router periodically checks mode flag of its neighbors to decide whether to work with deterministic or adaptive routing. According to [31] if at least one of the neighboring routers were congested the router would decide to work with adaptive routing; otherwise it would work with deterministic routing. To model a DyAD router in Rebeca we add a mode flag to our model [15] . The mode flag becomes true if the size of the corresponding input buffer reaches the congestion threshold.

Analyzing NoC Design, and Evaluating Different Routing Algorithms. Timing analysis of NoCs is required to discover possible deadline misses for packets traveling through the network. Based on the results of such analysis, suitable design decisions can be made. In asynchronous systems, lack of a reference clock leads to an interleaved execution of processes. Therefore, in GALS NoCs, a sent packet might be delayed by different numbers of disrupting packets and may have various end-to-end latencies. For analysis of such systems, it is essential to consider all possible behaviors of the system rather than specific traces.

The timed version of ASPIN is modeled and analyzed in [13] using simulation and model checking. Afra toolset was used for checking deadlock freedom, and message arrival, and for estimating the maximum end-to-end packet latency in the model. In the Rebeca model, we considered hardware features like switching strategy, communication protocols, and buffer and link delays. Packet latencies are computed with different design parameters, specially buffer sizes. Different routing algorithms are analyzed and compared.

The model is validated through comparing the extracted results to that of HSPICE [32], under both manual and real traffics [13]. Note that in HSPICE simulator, the lowest level of simulation in hardware domain is performed, and all the details of transistors and wires are considered.

Scalability Challenges. Clearly we cannot generate all the possible scenarios of packet injection in the network. We use PARSEC benchmarks [33] for choosing our scenarios. PARSEC is a well-known set of scenarios for packet generation in network on chip. For performance estimation, the Black-Scholes scenario from this benchmark has been selected.

For estimating maximum end-to-end packet latency, in order to analyze large NoCs, an scalable approach is proposed based on compositional verification

[13]. The compositional approach is specific for the XY-routing algorithm. The method computes the maximum end-to-end latency in GALS NoCs with XY routing algorithm in two steps. It breaks the path of a packet to its destination into horizontal and vertical sub-paths and then performs latency estimation in each sub-path separately. At the end, the results for each sub-path are combined to get latency estimation of the whole path. To do so, possible paths for each packet should be investigated precisely to find out which packets may make disruption for the transferring packet. To check the correctness of the method, these disruptions are considered in the scenarios and then the results are compared to that of HSPICE.

6 Discussion

Here we discuss the points raised in the introduction section, mainly faithfulness, usability, and analyzability.

Faithfulness. When we make a Rebeca model of a given system (or based on a given specification), first we want to know the set of actors that build the model. We start by finding the modules that are running concurrently in the system and communicate asynchronously via message passing. Each of these modules will be represented by an actor in the model. Each actor may represent a module that may contain different sub-modules that are not executed concurrently, or are communicating synchronously (like in Globally Asynchronous, Locally Synchronous systems). Networks of nodes which communicate through asynchronous messages build systems with a model of computation perfectly matching Rebeca. This is the case for all the three examples provided in Sections 3 to 5.

The level of abstraction in modeling depends on the properties of the model we are interested in. For different aspects to be checked we have to model different features of the system.

Usability. As for usability, we focused on effectiveness, efficiency, and satisfaction. Rebeca is Usable for software engineers and programmers. They are familiar with the Java-like syntax of Rebeca, and with the object-oriented style of programming. For concurrent programming, programmers are mostly using thread-based programming, and the event-based model of computation may not be as widely used by all the programmers. Usually it would be enough to tell them that each actors is one thread of execution, and message servers run atomically with no preemption. To be completely fair, it is worth mentioning that designing the code with an event-driven style may not be straight forward for all the programmers, but it is learned fairly quickly. Hardware and electrical engineers are more familiar with event-driven computation. But based on our observation, electrical engineers prefer a component-based system, like what they get with Simulink.

Reusability and Design Patterns. In Sections 3, 4, and 5 we have subsections on how the Rebeca code can be reused or extended for similar applications in the same domain. Based on our experience on the NoC design, we proposed a generic pattern for track-based traffic control systems and used it for building a coordinated actor model for adaptive air traffic control systems [34]. We used Timed Rebeca in modeling mobile agents, using this pattern, but different in the analysis part. We came up with a light-weight approach in planning using this model [35].

Analyzability and Synthesis. Based on the asynchrony and isolation of actors, we designed specialized reduction techniques in model checking Rebeca and Timed Rebeca. In some cases, like for analyzing SystemC codes, we needed to extend Rebeca to have wait statements and global variables [36]. In these cases the MoC is no more the same, and most of our reduction techniques will no more work.

So far, synthesis has not been the focus of our research. But in the cases that Rebeca models represent the network protocol, then the implementation of the protocol can be just a refinement of the Rebeca code.

Traceability and Compositionality. Isolated units of concurrency makes the model modular, also effective compositional verification techniques are introduced. But there is no compositional semantics for Rebeca, mainly because of the message buffers. Traceability is high between the model and the system, but at the level of semantics and transition systems, we are dealing with similar problems like for other modeling languages.

Expressiveness and Rebeca Extensions. The discussion in this paper is based on the assumption that we have a language that is expressive enough for the domain of our interest. We had to extend Rebeca to increase its expressiveness where necessary. Timed and probabilistic extensions of Rebeca were introduced because the expressive power of Rebeca was not enough to capture the notions of time and probability. An ongoing project is extending Rebeca to model cyber-physical systems by supporting actors with continuous behavior, and for that we need the capability of defining linear differential equations. Different extensions of Rebeca build an actor family of languages [37].

Future Trends. Modeling cyber-physical systems using an extension of Rebeca, and building analysis techniques for this domain is a current ongoing project. Rebeca supports dynamic creation and topology in theory, but in none of the techniques we have carefully considered this dynamicity. Recently, the possibility of passing rebec names and hence having dynamic topology is added to the Rebeca tools. This is mostly necessary for modeling and analyzing autonomous and self-adaptive systems which are another domain of interest. For the techniques, in the future, we plan to focus more on synthesis, and also testing.

Acknowledgements

My work is supported in part by DPAC Project (Dependable Platforms for Autonomous Systems and Control) at Mälardalen University, Sweden, and by the project Self-Adaptive Actors: SEADA (nr 163205-051) of the Icelandic Research Fund.

I would like to thank the Rebeca team, everyone who worked in the team since 2001, and helped us to gain the insight and the experience, and be where we are now. Special thanks go to Ehsan Khamespanah for refactoring the model checking tool of Rebeca and extending and maintaining it for years. For writing this paper I had to bug the leaders of the three projects explained in the three sections, 3, 4, and 5; many thanks to Ehsan Khamespanah, Fatemeh Ghassemi, and Zeinab Sharifi. I would also like to thank Reinhard Wilhelm for challenging me on the explanation of the main concepts discussed in this paper. The discussion significantly improved the paper. Thanks to Hossein Hojjat for his careful comments throughout the paper, thanks to Paolo Masci for reading the paper and giving me the right pointers for the standard definitions of usability, thanks to Mohammad Reza Mousavi who kept me on my toes when talking about expressiveness, thanks to Tom Henzinger for his reassuring words, thanks to Alessandro Papadopoulos whose comments made me add some final notes to the paper, and thanks to Hans Hansson, Jan Friso Groot, Mohsen Vakilian and Amin Shali for finding the paper thought provoking and enjoyable. Last but not least thanks to Edward for inspiring me and being the drive and support to finally write this paper.

References

1. Manna, Z., Pnueli, A. In: On the faithfulness of formal models. Springer Berlin Heidelberg, Berlin, Heidelberg (1991) 28–42
2. Ptolemaeus, C., ed.: System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014)
3. ISO: Ergonomics of human-system interaction part 210: Human-centred design for interactive systems. Technical Report ISO 9241-210:2010, International Organization for Standardization (2010)
4. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, USA (1990)
5. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.: Modeling and verification of reactive systems using Rebeca. Fundamenta Informatica **63**(4) (Dec. 2004) 385–410
6. Sirjani, M.: Rebeca: Theory, Applications, and Tools. In: Formal Methods for Components and Objects, FMCO 2006, LNCS 4709, Springer (2006) 102–126
7. Sirjani, M., Jaghoori, M.M.: Ten years of analyzing actors: Rebeca experience. In: Formal Modeling: Actors, Open Systems, Biological Systems, LNCS 7000, Springer (2011) 20–56
8. Sirjani, M., Khamespanah, E.: On time actors. In: Essays Dedicated to Frank De Boer on Theory and Practice of Formal Methods - LNCS 9660, Springer (2016) 373–392
9. Khamespanah, E., Mechitov, K., Sirjani, M., Agha, G.A.: Schedulability Analysis of Distributed Real-Time Sensor Network Applications Using Actor-Based Model Checking. In: 23rd International Symposium on Model Checking Software, SPIN 2016, LNCS 9641, Springer (2016) 165–181
10. Khamespanah, E., Mechitov, K., Sirjani, M., Agha, G.A.: Modeling and Analyzing Real-Time Wireless Sensor and Actuator Networks Using Actors and Model Checking. In: Software Tools for Technology Transfer. (2017)

11. Yousefi, B., Ghassemi, F., Khosravi, R.: Modeling and Efficient Verification of Broadcasting Actors. In: 6th International Conference Fundamentals of Software Engineering, FSEN 2015, LNCS 9392, Springer (2015) 69–83
12. Yousefi, B., Ghassemi, F., Khosravi, R.: Modeling and Efficient Verification of Wireless Ad-hoc Networks. Formal Aspects of Computing (6) (2017) 1051–1086
13. Sharifi, Z., Mosaffa, M., Mohammadi, S., Sirjani, M.: Functional and performance analysis of network-on-chips using actor-based modeling and formal verification. ECEASST, AVoCS 2013 Proceedings **66** (2013)
14. Sharifi, Z., Mosaffa, M., Mohammadi, S., Sirjani, M.: Performance analysis of gals noc using actor models. Draft (2017)
15. Sharifi, Z., Mohammadi, S., Sirjani, M.: Comparison of NoC routing algorithms using formal methods. In: Proceedings of PDPTA 2013. (2013)
16. Hewitt, C.: Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. MIT artificial intelligence technical report (1972)
17. Aceto, L., Cimini, M., Ingólfsdóttir, A., Reynisson, A.H., Sigurdarson, S.H., Sirjani, M.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. In: FOCLASA. (2011) 1–19
18. Reynisson, A.H., Sirjani, M., Aceto, L., Cimini, M., Jafari, A., Ingólfsdóttir, A., Sigurdarson, S.H.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. Sci. Comput. Program. **89** (2014) 41–68
19. Khamespanah, E., Sirjani, M., Sabahi-Kaviani, Z., Khosravi, R., Izadi, M.: Timed rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. Science of Computer Programming **98** (2015) 184–204
20. Akyildiz, I., Vuran, M.C.: Wireless Sensor Networks. John Wiley & Sons, Inc., New York, NY, USA (2010)
21. TinyOS: TinyOS community forum: An open-source OS for the networked sensor regime Available through http://www.tinyos.net.
22. Khamespanah, E., Khosravi, R., Sirjani, M.: An efficient tctl model checking algorithm and a reduction technique for verification of timed actor models. In: Science of Computer Programming. (2017)
23. El-Hoiydi, A.: Spatial TDMA and CSMA with preamble sampling for low power ad hoc wireless sensor networks. In: Proceedings of the Seventh IEEE Symposium on Computers and Communications (ISCC 2002). (2002) 685–692
24. Polastre, J., Hill, J.L., Culler, D.E.: Versatile low power media access for wireless sensor networks. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys 2004. (2004) 95–107
25. Yousefi, B., Ghassemi, F.: An efficient loop-free version of aodvv2. CoRR **abs/1709.01786v2** (2017)
26. Ghassemi, F., Fokkink, W.: Model Checking Mobile Ad-Hoc Networks. Formal Methods in System Design **49**(3) (2016) 159–189
27. Benini, L., Micheli, G.D.: Networks on Chips: A New SoC Paradigm. IEEE Computer **35**(1) (2002) 70–78
28. Guerrier, P., Greiner, A.: A Generic Architecture for On-Chip Packet-Switched Interconnections. In: 2000 Design, Automation and Test in Europe (DATE 2000). (2000) 250–256
29. Sheibanyrad, A., Greiner, A., Panades, I.M.: Multisynchronous and Fully Asynchronous NoCs for GALS Architectures. IEEE Design & Test of Computers **25**(6) (2008) 572–580
30. Chiu, G.M.: The Odd-Even Turn Model for Adaptive Routing. IEEE Trans. Parallel Distrib. Syst. **11**(7) (July 2000) 729–738

31. Hu, J., Marculescu, R.: DyAD: Smart Routing for Networks-on-chip. In: Proceedings of the 41st Annual Design Automation Conference. DAC '04 (2004)
32. HSPICE: HSPICE Homepage: https://www.synopsys.com/verification/ams-verification/circuit-simulation/hspice.html
33. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC Benchmark Suite: Characterization and Architectural Implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. PACT '08 (2008)
34. Bagheri, M., Akkaya, I., Khamespanah, E., Khakpour, N., Sirjani, M., Movaghar, A., Lee, E.A.: Coordinated actors for reliable self-adaptive systems. In: 13th International Conference on Formal Aspects of Component Software, FACS 2016. (2016)
35. Castagnari, C., de Berardinis, J., Forcina, G., Jafari, A., Sirjani, M.: Lightweight preprocessing for agent-based simulation of smart mobility initiatives. In: FOCLASA 2017 Proceedings. (2017)
36. Razavi, N., Behjati, R., Sabouri, H., Khamespanah, E., Shali, A., Sirjani, M.: Sysfier: Actor-based formal verification of SystemC. ACM Transactions on Embedded Computing Systems (2009)
37. de Boer, F.S., et al: A survey of active object languages. ACM Comput. Surv. **50**(5) (2017) 76:1–76:39