Ten Years of Analyzing Actors: Rebeca Experience

Marjan Sirjani[1,2], Mohammad Mahdi Jaghoori[3]

[1] School of Computer Science, Reykjavik University, Reykjavik, Iceland
[2] University of Tehran, Tehran, Iran,
[3] CWI, Amsterdam, The Netherlands
marjan@ru.is, jaghouri@cwi.nl

Abstract. In this paper, we provide a survey of the different analysis techniques that are provided for the modeling language Rebeca. Rebeca is designed as an imperative actor-based language with the goal of providing an easy to use language for modeling concurrent and distributed systems, with formal verification support. Throughout the paper the language Rebeca and the supporting model checking tools are explained. Abstraction and compositional verification, as well as state-based reduction techniques including symmetry, partial order reduction, and slicing of Rebeca are discussed. We give an overview of a few extensions of Rebeca. For example, we present the modular schedulability analysis of timed actor-based models and formal techniques to check correctness of self-adaptive systems using Rebeca. A summary of design decisions and a brief general comparison of the analysis methods are provided at the end of the paper while specific sections are accompanied with examples and corresponding related work.
keywords: Actors, Rebeca, Concurrency, Formal Verification, Model Checking, Reduction Techniques, Abstraction.

1 Introduction

As information networks are becoming increasingly important in our society, the number of distributed heterogeneous software systems is rapidly growing. Distributed systems consist of multiple cooperating components where the components are typically encapsulated systems or objects spread over a network, interacting via asynchronous communication. Web-service applications and applications based on wireless network technologies are examples of such distributed and asynchronous applications. Such technologies are now used in a vast variety of applications, such as medical systems, transportation systems, and the significant business of online gaming.

The actor model is among the pioneering ones to address concurrent and distributed applications. The actor language was originally introduced by Hewitt [44] as an agent-based language for programming distributed systems, and was later developed by Agha [3, 6, 5] into a concurrent object-based model. Valuable work has been done on formalizing the actor model by Talcott et al. [5, 81,

113, 114]. The actor model has been used both as a framework for theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent and distributed systems.

In the actor model, *actors* are the universal primitives of concurrent computation: in response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message that it receives. Actors have encapsulated states and behavior, and are capable of creating new actors and redirecting communication links through exchange of actor identities.

Different interpretations, dialects and extensions of the actor model are proposed in several domains, for example, designing embedded systems [78] and wireless sensor networks [19]. The actor model is further claimed to be the suitable model of computation for the most dominating application domains of multi-core programming and web services [45, 17, 16]. Compared to mathematical modeling languages, like process algebras, actors are more natural for designers, software engineers, modelers and programmers. Compared to process-oriented models, like Petri nets, the actor model has the advantages of an object-based language, like encapsulation of data and process, and more decoupled modules. Moreover, the formal semantics of actor-based languages builds a firm foundation for formal analysis and verification.

The actor model of concurrency is getting more and more popular in practice [64, 46, 47]; as a few examples Erlang [37] and Scala [98] are two programming languages that have applied the actor model of concurrency and are now getting widely used. The Asynchronous Agents Library is an actor-based framework that is added to Microsoft Visual Studio 2010 [85]. Actors have been used in real-world applications like Twitter's message queuing system, and Vendetta's game engine (in 2010) [76].

Applying formal methods in software engineering is an important step towards building more reliable and robust systems. In the more than twenty-five years since its invention, model checking has achieved multiple breakthroughs, bridging the gap between theoretical computer science and practical computer engineering. Today, model checking is extensively used in the hardware industry, and has also become feasible for verification of many types of software [23]. State space explosion is the dominant problem for model checking; investigating new abstraction and reduction techniques is the leading edge in this research area.

Reactive Objects Language, *Rebeca* [102, 106], is an operational interpretation of the actor model with formal semantics and model checking tools. In a Rebeca model, system components are sets of reactive objects, called *rebecs*, which communicate with each other and with their environment, through message passing. Message passing is asynchronous and fair. Messages related to each rebec are stored in the message queue of the rebec. The computation takes place by taking the message at the head of the message queue and executing the corresponding message server [106].

Ten years ago we noticed the urgent need for a formal tool that could be easily used by software engineers. By observing the increasing number of con-

current and distributed systems on one hand, and popularity and efficacy of object-oriented approaches on the other hand, we identified the actor model as the best candidate to be the basis of our research. So, Rebeca was designed to bridge the gap between formal methods and software engineering. In Rebeca, reactive objects are units of concurrency. Compared to thread-based concurrent programming, Rebeca makes the modeling of concurrency more natural and less error-prone. Moreover, it brings transparency by removing the difference between local and non-local concurrency in a distributed system. The recent widespread use of actors in different areas and applications, like in distributed applications, or safe/sound programming of emerging multicore hardware, clearly demonstrates its power as a computational model [45, 47].

To the best of our knowledge, the first attempt to provide compositional verification and model checking support for an imperative actor-based language is the introduction of Rebeca in 2001 [102–104]. We defined the language Rebeca and its formal semantics, developed its model checking tools, and provided a compositional verification theory and abstraction techniques. We have been actively and successfully investigating specialized reduction techniques for formal verification of Rebeca models, namely, symmetry, partial order, and slicing, that are all based on the formal semantics of the language [107, 105, 108, 58, 56, 48, 97, 96]. Rebeca with its simple message-driven object-based computational model, Java-like syntax, and the associated set of verification tools, is an interesting and easy-to-learn model for students, software engineers, and practitioners.

In this paper, we present a survey of the different analysis techniques that are developed for Rebeca. We[4] first started model checking using back-end model checking tools of Spin [111] and NuSMV [88], and then proceeded to develop direct model checking tools (Section 3). To tackle the state space explosion problem, we developed a theory for abstraction and compositional verification of Rebeca models (Section 4). Then we moved towards investigating and applying reduction techniques in model checking, including symmetry and partial order reduction (Section 5), and slicing (Section 6). Our focus has been on finding specific reduction techniques that exploit the specific features of Rebeca models. In our techniques for symmetry reduction, partial order reduction, slicing and distributed model checking, we perform static analysis on the model and use the results to tackle the state space explosion problem.

The language Rebeca is used in different research and application areas. We established schedulability analysis for real-time actor-based models and used Rebeca to represent our approach (Section 7). We have also extended Rebeca to serve as a policy-based coordination language in modeling self-adaptive systems, and we have established corresponding techniques to check their correctness (Section 8).

[4] Please note that the word *we* in this paper refers to all those who have contributed and are working on designing and developing Rebeca, and the related theories, tools, applications, and extensions.

2 Rebeca Syntax and Semantics

Rebeca [104–106] is a modeling language with a formal semantics based on an operational interpretation of the actor model [44, 6]. The definition of a Rebeca model consists of a set of reactive classes plus an initial configuration in its **main** section where a set of *rebecs* (reactive objects) are created as instances of reactive classes (see Fig. 1 for the syntax of Rebeca). Each rebec has a single thread of execution. The behavior of a Rebeca model is defined by the fair and interleaved execution of the rebecs.

Rebecs communicate *only* through asynchronous message passing and have unbounded buffers for automatically storing the incoming messages, i.e., there is no statement in Rebeca syntax to explicitly wait for receiving a message. When a rebec is scheduled to run, the message at the head of the queue is taken out and processed. Each message that can be serviced by a rebec has a corresponding message server, which is given in the definition of the reactive class denoting the type of the rebec. Message servers are executed atomically, thus, Rebeca is said to have coarse-grained ("big-step") interleaving. Although each rebec has one queue, we can model multiple reception queues between different rebecs by adding an extra rebec to represent each reception queue.

A **reactiveclass** definition takes an integer argument to denote an upper-bound on the length of the message queue; this is used in model checking and can be increased in the case of a queue overflow. The body of the **reactiveclass** consists of three parts: the known rebecs section includes a set of rebec identifiers and as such forms the initial communication topology of the system; variables constitute the local state of a rebec; and, message servers (also called methods) define the behavior. Each message server may declare local variables and further contains a sequence of statements, including assignments, if statements, rebec creation (**new**), and method calls. A method call is equivalent to sending an asynchronous message that invokes the corresponding message server (method). By sending rebec variables around (i.e., the variables holding a rebec identifier),

$$
\begin{array}{lll}
\mathit{Model} & ::= \mathit{Class}^*\ \mathit{Main} & \mathit{Stmt} ::= v = e; \mid v = \textbf{new}\ C(\langle e \rangle_,^*); \\
\mathit{Class} & ::= \textbf{reactiveclass}\ C(\mathit{Nat}) & \quad \mid \mathit{Call}(\langle e \rangle_,^*); \\
 & \quad \{\mathit{KRs}\ \mathit{Vars}\ \mathit{MsgSrv}^*\} & \quad \mid \textbf{if}\ (e)\ \mathit{MSt}\ [\ \textbf{else}\ \mathit{MSt}\ ] \\
\mathit{KRs} & ::= \textbf{knownrebecs}\ \{\ \langle \mathit{Vdcl}; \rangle^*\ \} & \mathit{Call} ::= v.M \mid \textbf{self}.M \mid \textbf{sender}.M \\
\mathit{Vars} & ::= \textbf{statevars}\ \{\ \langle \mathit{Vdcl}; \rangle^*\ \} & \mathit{Mst} ::= \{\mathit{Stmt}^*\} \mid \mathit{Stmt} \\
\mathit{Vdcl} & ::= T\ \langle v \rangle_,^+ & \mathit{Main} ::= \textbf{main}\ \{\ \mathit{Reb}^*\ \} \\
\mathit{MsgSrv} & ::= \textbf{msgsrv}\ M(\langle T\ v \rangle_,^*)\ \{\mathit{Stmt}^*\} & \mathit{Reb} ::= T\ r(\langle T\ r \rangle^*) : (\langle T\ e \rangle^*);
\end{array}
$$

Fig. 1. BNF grammar for Rebeca classes. Angle brackets $\langle ... \rangle$ are used as meta parentheses, square brackets [...] for optional parts, superscript + for repetition more than once, superscript * for repetition zero or more times, whereas using $\langle ... \rangle_,$ with repetition denotes a comma separated list. Identifiers C, T, M, r and v denote class, type, message server, rebec and variable names, respectively; *Nat* denotes a natural number; and, e denotes an (arithmetic, boolean or nondeterministic choice) expression.

```
1   reactiveclass Sender(4) {
2   knownrebecs {
3       Receiver r;
4   }
5   statevars {
6       int req;
7       boolean pass;
8   }
9   msgsrv initial() {
10      req = 1;
11      r.receiveReq(req);
12  }
13  msgsrv sendNextReq() {
14      pass = ?(true,false);
15      if(pass) req = req + 1;
16      if(req == 5) req = 1;
17      r.receiveReq(req);
18  }
19 }
20  main() {
21      Sender s(r):();
22      Receiver r(s):();
23  }
```

```
1'  reactiveclass Receiver(4) {
2'  knownrebecs {
3'      Sender s;
4'  }
5'  statevars {
6'      int msg;
7'      boolean isFinal;
8'  }
9'  msgsrv initial() {
10'     isFinal = false;
11' }
12' msgsrv receiveReq(int m) {
13'     msg = m;
14'     if(msg == 4)
15'         isFinal = true;
16'     else
17'         isFinal = false;
18'     s.sendNextReq();
19' }
20' }
```

Fig. 2. The Rebeca code of the sender/receiver example

the topology can change dynamically. In each reactive class, there is at least one message server, called ' initial '; this is responsible for initialization tasks (like 'constructors' in object oriented programming languages). Each rebec receives this message implicitly upon creation. The system continues running as long as there is at least one message to be processed. To instantiate a **reactiveclass** in the **main** section, one should provide first the bindings of the **knownrebecs** and then the parameters to the initial message server (if any).

2.1 A Sender/Receiver Example

As a running example, we use a simple model of a sender and receiver (shown in Fig. 2). We use slightly different versions of this example in different sections to demonstrate how the techniques in that section can be used in practice. This example is similar to the *alternating bit protocol*, but we simplified it by putting a nondeterministic assignment (line 14) instead of getting a real acknowledgment from the receiver.

There is a rebec in this example that acts as a sender and sends a number of messages (four here) to a receiver (the other rebec). Based on the nondeterministically chosen value of the variable pass (line 14), the sender rebec either sends a new message (line 15) or repeats the previous one. The happy scenario is when the receiver receives all four messages, after which isFinal, a state variable of the receiver, is set to *true* (line 15'). Every time receiveReq is executed by the receiver, a sendNextReq is sent back to the sender asking for the next message (line 18'). After receiving the last message by the receiver this scenario starts over again.

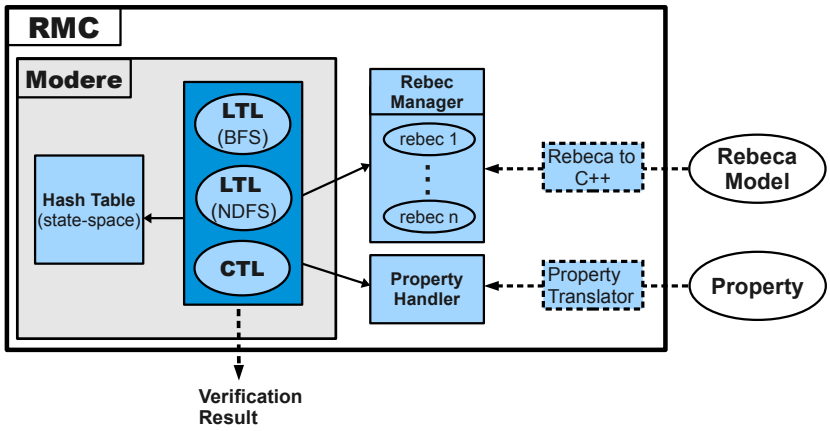

Fig. 3. The architecture of Rebeca Model Checker (RMC) — the solid arrows show calling another component (control dependencies), and dashed arrows show flow of data (input/output of the tool and the data dependencies)

A possible interesting property for this example is *G(F(isFinal == true))* which checks whether the last message is always finally received by the receiver. The property is an LTL (Linear Temporal Logic) formula [35] where G denotes *globally* and F denotes *finally*. In addition, the example can be verified against deadlock. Deadlock occurs for example if we remove line 18' from the model.

3 Model Checking Tools

Since 2005, Rebeca has a custom-made explicit-state model checker [56, 58] implemented in C++. The advantage of a tailor-made tool is that it can take into account the intrinsic features and the nature of the concurrency model of Rebeca. This amounts to a more efficient tool, provided that it is debugged as any other software and it is maintained and kept up-to-date with respect to the state of the art in tools and algorithms. However, model checking Rebeca has been possible since 2003; before the development of the custom model checking tool, this was achieved by translation into the input languages of famous and prominent model checkers [103, 110, 107, 48], namely, SMV [20], Promela (SPIN) [49] and later mCRL2 [42]. The main advantage of using such translations is that we can benefit from the maturity of the back-ends of established tools. In this section, we give an overview of the tools developed for model checking Rebeca directly.

Fig. 3 shows the architecture of Rebeca Model Checker (RMC) which is composed of loosely coupled components. The main component is *Modere*, the Model-checking Engine of Rebeca. A model checking engine is in essence a highly optimized and memory efficient search algorithm, e.g., DFS or BFS, which is extended to check for temporal properties specified, e.g., in LTL or CTL (Compu-

tational Tree Logic) [35]. Modere was first developed as an LTL model checker; Vardi [118] argues that LTL as a linear-time temporal logic is more expressive and intuitive and supports compositional reasoning, unlike CTL as a branching time temporal logic. Nevertheless, CTL is advantageous, for example, in specifying reachability goals. Modere can now check for properties in both LTL and CTL, thus, it can fulfill the needs of a wide range of practitioners. It also has implementations of both DFS and BFS based search algorithms. The modular architecture of the RMC tool-set has allowed us to make these extensions to Modere while reusing the other components, e.g., Rebeca to C++ converter.

The rebec manager component in RMC is model dependent and is created by a Rebeca to C++ translator (see Fig. 3), it contains a C++ equivalent of the reactive class definitions and the instantiation of these classes as specified in the Rebeca model. During model checking, Modere repeatedly puts the rebecs in a specific state and asks the rebec manager to run one rebec, the rebec manager returns the resulting state(s) afterwards. A possible extension point of the tool-set is to replace the rebec manager with a process/object manager for another language with a similar actor-based computational model.

3.1 Bounded Model Checking.

Bounded model checking has also been studied for Rebeca using the SMT (Satisfiability Modulo Theories) solver tool Yices [33]. Such a tool checks the satisfiability of a given formula; this is in theory NP-hard, but there are several efficient heuristics for this problem that work in practice and many tools have been developed. This approach is particularly useful to model check Rebeca for data-centric applications.

To use an SMT-solver, first the general concurrency and communication model of Rebeca needs to be defined as a set of formulas [59]. And then given a specific model to be verified, the model needs to be translated into compatible formulas. Finally, the desired property is also turned into formulas, such that the conjunction of these three sets of formulas: "$RebecaConcurrency \wedge RebecaModel \wedge Property$" is satisfiable if and only if the given property holds for the given model.

Table 1. Defining Rebeca concepts as formulas

Rebeca Concept	type	description
(isActiveRebec i t)	bool	Rebec i is active at step t
(isActiveMsgsrv j i t)	bool	Message server j of rebec i is active at step t
(qSize i t)	int	The size of queue of rebec i at step t
(queue i j t)	int	Element i in queue of rebec j at step t.

To encode the general concurrency model of Rebeca, we first need to model the basic concepts of a Rebeca model. A possible encoding for this is shown in Table 1. In this encoding, all messages are represented as integers (cf. the type

of queue elements in Table 1). The parameter t in this encoding models the execution steps; for example, to send a message at step t to rebec i, we need to make sure that it exists in the queue of rebec i at step $t+1$ and furthermore, it should hold that (qSize i $t+1$) = (qSize i t) + 1.

As the next step, to encode a specific model, we need to set up some constants, like the number of rebecs in the system, as well as translate the class and message server definitions. The latter is achieved again by defining the changes that need to take place as a transition from step t to step $t+1$.

Encoding temporal properties, e.g., LTL, is done by unfolding the property. For example a property Gp is translated to $p(0) \wedge \ldots \wedge p(k)$ where $p(t)$ asserts the satisfiability of p at step t and k is the bound we assume on the number of steps in bounded model checking. If a counter-example is obtained with this bound, we know that the desired property does not hold for the system. But the satisfiability of the desired property up to k steps cannot guarantee the correctness of the model in general. One can increase this bound to obtain more general results as far as the physical (memory and time) restrictions of the hardware allow it.

3.2 Domain Specific Model Checking: SystemC Designs

SystemC [89] is an object-oriented language that has emerged lately as the leading language for system-level modeling. In the project Sysfier [2], a tool is integrated into Rebeca model checking tools to map SystemC designs to Rebeca models and then use Rebeca verification tool-set to verify LTL and CTL properties [10, 92]. Many examples are translated from SystemC to Rebeca and are model checked against LTL and CTL properties.

To model SystemC designs, Rebeca is extended by adding global variables and wait statements. Global variables are used in a very limited way to model events and signals. A *wait* statement is used when a process needs to wait for a specific event before it can continue. The simulation kernel of SystemC is mapped to a rebec which plays the role of a synchronizer.

The inherent similarities of the two languages prevents any unwanted overhead or additional states. As a matter of fact, the theories in abstraction and compositional verification of Rebeca and the tools and techniques for state space reduction can be applied to SystemC verification, too. Moreover, the Rebeca model checkers are equipped with different policies based on the semantics of SystemC to reduce the state space. One policy that considerably reduces the size of the generated state space is to mimic the behavior of the SystemC simulation kernel and consider the sequential execution of rebecs instead of all possible interleavings. This policy works when verifying race-free SystemC designs. Another policy is to apply partial order reduction based on SystemC semantics.

4 Compositional Verification and Abstraction

In a broad sense, compositional verification tackles the state-space explosion problem by verifying the constituent components of a system in isolation: since these components are smaller in size, they are more amenable to computerized analysis, e.g., model checking. The correctness properties of the system are then derived from the properties of its individual components [22, 73, 83].

In general, compositional verification may be exploited more effectively when the model is naturally decomposable [95]. Actor-based models provide such inherently independent modules because there are no shared variables, but explicit non-blocking send operations are the only way of communication.

In compositional verification of Rebeca [106], we follow a top-down approach, where components are sub-models and are the result of decomposing a closed model. To this end, we take one part of the closed model as an open component and the rest of the model is assumed to be the environment. With such decomposition, the rebecs in the selected component will be modeled with their state and behavior, whereas the state-space of the *external* rebecs, i.e., those in the environment, is not modeled because their methods are not executed. External rebecs are only modeled as their potential in sending messages.

In an unrestricted environment, the general so-called *environment problem* arises, which states that the reachable state space of an open component, as described above, may in fact be much larger than that of the original closed model. In fact, putting the messages sent from external rebecs into the queues will immediately overflow the queues. To alleviate this problem, we model a reduced environment which can be considered as a *compositional minimization*. To do so, we consider the set of external messages always enabled, and consider a fair choice between executing these messages and the message on the top of the queue. This way we also avoid explicitly modeling the environment, for example, as another rebec.

In order to prove certain properties, we may need some assumptions about the environment, but in general we do not apply assume/guarantee reasoning. Our compositional verification generates an over-approximation of the behavior of the components. We proved a weak simulation relation between any closed model including the component and the component composed with the abovementioned environment, and hence we can claim that safety properties are preserved [106].

Another view to compositional verification is a bottom-up approach. In [108], we discuss such an approach along the lines of modular verification where the concept of a component is an independent module with a well-defined interface. Such notion of a component represents a re-usable off-the-shelf module. Once verified, this module has a fixed proven specification and then it can be used to build reliable systems. In [108], this modular verification technique for Rebeca is presented. In the both approaches described above, although the strategy in abstraction techniques is the same, the technical details (to keep the theory valid) are quite different.

5 Symmetry and Partial Order Reduction

Two of the most widely used state-space reduction techniques in explicit-state model checking are symmetry [36, 51, 25, 86] and partial order reduction [41, 117]. These two are the first reduction techniques implemented in the model checking engine of Rebeca (Modere) [58, 56] because they fit naturally the asynchronous object model of Rebeca and yield reasonable reductions.

5.1 Partial Order Reduction

The idea of partial order reduction (POR) is that it is not always necessary to consider all of the possible interleaved sequences of the enabled actions. Instead, the execution of some of them can be postponed to a future state without affecting the validity of the correctness property. This way, the full interleaving of those actions is avoided and the size of the explored state space is reduced.

A popular approach to implementing POR is based on statically detecting *safe* actions, called static POR. This approach is applicable to Rebeca only in absence of dynamic rebec creation or change of topology. The characteristics of safe actions are described in the following. The first characteristic is invisibility, i.e., not changing the satisfiability of the correctness property. In the sender/receiver example in Fig. 2, assume the correctness property is *G(F(isFinal == true))*. The message server sendNextReq is then invisible as it does not change the variable isFinal. In addition, the initial message servers are also by definition invisible [56], because variables are uninitialized before that.

Two actions are said to be independent if one cannot disable the other; an action that is independent of all other actions is called globally independent. Due to absence of shared variables, assignments are local and hence globally independent; therefore, independence of a message server depends only on its send statements (recall that no **new** statement is allowed in static POR). Sending a message from $r1$ to $r2$ is globally independent if $r1$ is the only rebec that may send messages to $r2$ [56]. In the sender/receiver example, since only s sends messages to r and vice versa, i.e., there are even no self calls, all rebec actions in this example are globally independent.

An action is safe if it is safe and globally independent. The model checker can execute a safe action without considering its interleaving with other actions. The initial message servers as well as sendNextReq correspond to safe actions in our example.

Considering the coarse-grained interleaving of Rebeca, an action corresponds to a message server; therefore, POR amounts to a considerable reduction when applicable. Furthermore, as mentioned above the initial message server is always invisible. This makes direct application of POR in Modere more efficient than translating for example to Promela where you will lose such useful information.

5.2 Symmetry Reduction

The symmetry reduction technique views the state space as a graph: the states are the vertices and the transitions are the edges. The idea then is to partition

```
1  reactiveclass SenderReceiver(4){
2    knownrebecs {
3      SenderReceiver peer;
4    }
5    statevars {
6      int req;
7      boolean pass;
8      int msg;
9      boolean isFinal;
10   }
11   msgsrv initial() {
12     req = 1;
13     peer.receiveReq(req);
14     isFinal = false;
15   }
16   msgsrv sendNextReq() {
17     pass = ?(true,false);
18     if(pass) req = req + 1;
19     if(req == 5) req = 1;
20     peer.receiveReq(req);
21   }
22   msgsrv receiveReq(int m) {
23     msg = m;
24     if(msg == 4)
25       isFinal = true;
26     else
27       isFinal = false;
28     sender.sendNextReq();
29   }
30 }
31 main() {
32   SenderReceiver sr1(sr2):();
33   SenderReceiver sr2(sr1):();
34 }
```

Fig. 4. A symmetric sender/receiver reactive class

the state space into equivalence classes corresponding to isomorphic graphs and use one state as the representative of each equivalence class. The problem in symmetry reduction is that calculating the representative states, known as the 'constructive orbit problem,' is NP-hard [24]. This is usually alleviated by first *specifying* or *detecting* the symmetry among higher-level constructs (such as processes or objects) using some static analysis and then applying it in solving the orbit problem. The most popular approach to explicitly specify symmetry in a system is the notion of scalar sets, proposed by Ip and Dill [51], which is also later used by others, e.g., [13], [43].

In Rebeca, since rebecs instantiated from the same reactive class exhibit similar behavior, any symmetry in the communication structure of a model leads to symmetry in the underlying state-space graph; we call this *inter-rebec symmetry*. Detecting such symmetries does not rely on any symmetry-related input from the modeler. In [58] we proposed a polynomial-time solution for detecting structural symmetry without requiring any change in the syntax. The sender/receiver example in Fig. 2 is not symmetric because the rebecs in the model are of different types. We revisit the example in this section by merging the two reactiveclasses thus enabling a rebec to act both as a sender and a receiver, shown in Fig. 4. The composition of the two rebecs, specified in the **main** section, is now symmetric, i.e., by swapping the names of the rebecs we obtain:

```
main {
  SenderReceiver sr2(sr1):();
  SenderReceiver sr1(sr2):();
}
```

which can be changed into the original bindings by reordering the lines. This can be contrasted to the example in Fig. 5, where there is a central receiver with three senders. In this case, swapping the names of the rebecs of type Sender does

```
1 reactiveclass Receiver(4) {
2   knownrebecs {
3     Sender s[i:1..3];
4   }
5   statevars {
6     int msg[i];
7     boolean isFinal[i];
8   }
9   ...
10 }
```

```
11 reactiveclass Sender(4) {
12   knownrebecs { Receiver r; }
13   ...
14 }
15 main{
16   Sender s1(r):();
17   Sender s2(r):();
18   Sender s3(r):();
19   Receiver r(s1,s2,s3):();
20 }
```

Fig. 5. A symmetric star topology.

not yield any symmetry, because this changes the knownrebecs binding of the Receiver rebec.

The example in Fig. 5 is still symmetric if we make sure the implementation of the Receiver class is internally symmetric. We extended our automatic symmetry detection further to also consider *intra-rebec* symmetries [57]. In intra-rebec symmetry, we propose to use scalar-sets but for a different purpose from its usual use, i.e., we use scalar sets locally for each class to specify the symmetric behavior of that class with respect to its known rebecs (when applicable), rather than specifying the symmetry in the whole system, as in e.g. [52, 14]. Our use of scalar sets is in line with the modular modeling encouraged by the actor model. This way we can consider the internal symmetry of rebecs along with the symmetry in their communication structure.

One of the topologies where intra-rebec symmetry can be used is a star network. Fig. 5 shows an example of sender and receiver reactive classes that are instantiated in a star topology. The definition of the Receiver class uses a scalar set i in defining the known rebecs. This implies that the definition of the reactiveclass must be symmetrical which is guaranteed by syntactical restrictions; for example, the state variables in this class definition are defined per known rebec. Rebeca compiler will check statically whether the model is symmetric and in that case, it will generate the necessary information for the model-checking engine (Modere) to apply symmetry reduction.

5.3 Applicable Properties

When using POR, the model checker may only consider Linear Temporal Logic without the next operator (LTL-X) [41, 117]; as we mentioned above, POR reduces the state space by postponing the execution of certain actions. This intuitively means that the 'next-state' behavior of the system is not preserved. When using symmetry, all temporal operators are allowed but the correctness property must also be symmetric. For example, with two instances of the SenderReceiver class (Fig. 4), the property asserting whether only sr1 can reach the goal, i.e., isFinal = true, is not symmetric; rather, we must check whether both sr1 and sr2 could reach this goal.

5.4 Related Work

The closest work to our inter-rebec symmetry detection is that of Donaldson, Miller and co-authors, who independently from our work, have proposed several techniques for detecting symmetry in similar models of computation (mainly Promela) [14, 29–32]. To be able to automatically detect symmetries in Promela, they need to assume a communication pattern like that of Rebeca, namely using static channels. Another similar work is by Leuschel and Massart [79] where there is no need to extend the syntax of B to specify symmetry, because a built-in construct, called *deferred sets*, gives rise to symmetric data values in a way similar to scalar sets. Symmetry detection still depends on the proper use of deferred sets by the modeler, whereas in Rebeca, symmetry detection is based on the intrinsic communication mechanism of the language.

Basset [77], is a general framework for testing actor systems compiled to Java bytecode. Basset employs heap symmetry reduction which is based on data symmetries and is thus orthogonal to the structural symmetry reduction technique that we have applied. On the other hand, Basset implements a dynamic POR technique based on the happens-before relation and the causality among message send and receive events. There is no need for such heavy-weight dynamic POR in Rebeca due to the stronger assumptions of coarse-grained interleaving and FIFO ordering of messages. Unlike in Modere, the reported implementation of Basset has not yet combined the use of symmetry and POR.

6 Slicing

Program slicing is a program analysis technique with applications in various software engineering activities such as program understanding, debugging, testing, program maintenance, and complexity measurement [87]. Slicing can be used together with model checking and is orthogonal to a number of other reduction techniques. In [34] a significant reduction is reported by slicing concurrent object-oriented source code, and slicing is recommended because of its automation and low computational cost. In this section, we review slicing of Rebeca codes and its specific features and data dependencies, which are reported in detail in [96] and [97].

Static slicing [120] extracts statements from a program which have a direct or indirect effect on computations of other statements. More specifically, a program slice consists of the parts of a program that potentially affect the values computed at some statement of interest (referred to as the slicing criterion). A slicing criterion is usually denoted by $\langle p,\ V\rangle$, where p is a program statement and V is a set of variables.

The main challenge in this area is to efficiently build a precise slice. One of the main approaches to slicing is using reachability analysis on a program dependence graph. A program dependence graph consists of nodes representing the statements of a program, and edges representing the control and data dependencies. There is a control dependency between two statement nodes if one

statement controls the execution of the other. Data dependency exists between two statement nodes if the change made to a variable at one statement might reach the usage of the same variable at the other statement. The general approach to slicing is to find all paths in the graph that affect a specific variable in a specific statement. We build all such paths, and hence the slice, by following the dependency edges backward from the given statement.

In a simple sequential program without procedures, the slicing method is trivial. Applying the same simple method to programs with procedures, may build paths that are not *realizable*. By realizable we mean those paths that show a possible execution of the program. In order to remove the unrealizable paths, we need a *context-sensitive* analysis: the computation of a slice must preserve the calling context of the called procedures, and ensure that only paths corresponding to the legal call-return sequences are considered. Context-sensitive slicing can be done by generating summary edges at call sites: summary edges represent the transitive dependence edges of called procedures at call sites [70].

In concurrent programs with shared variables another type of dependence arises: interference. Interference occurs when a variable is defined in one thread and used in a concurrently executing one. Like above, a simple traversal of interference dependence edges during slicing can produce unrealizable paths and make the slice imprecise. To solve this problem, we need to consider the valid execution chronology [71]. Considering the chronological order of statements within a thread is not enough to build precise slices. The reason is that it is in general not possible to determine whether a definition reaches the statement of interest (slicing criterion), or it is always *killed* (disabled) by other definitions. As a result, we may mistakenly consider the statement including that definition in the slice. So, the interference dependency is not transitive.

Slicing object-oriented programs presents new challenges which are not encountered in traditional program slicing [87]. To slice an object-oriented program, features such as classes, dynamic binding, encapsulation, inheritance, message passing and polymorphism need to be considered carefully. Although the concepts of inheritance and polymorphism are strengths of object-oriented programming languages, they pose special challenges in program slicing. Due to inheritance and dynamic binding in object-oriented programs, the process of tracing dependencies becomes more complex than that in a procedural program.

6.1 Slicing Rebeca

In [97, 96] we proposed slicing techniques for Rebeca which are based on Rebeca's actor-based computational model. We introduced a specialized control graph for Rebeca to capture its reactive behavior. One can perform data flow analysis on a Rebeca model by iterating over its control flow graph. Rebeca dependence graph is then introduced to represent different dependencies including control, data, intra-rebec, parameter-in, activation, member, and known-rebec dependencies. These two types of graphs are explained below.

A *Rebeca control flow graph (RCFG)* is defined based on the atomic execution of message servers and the reactive behavior of the rebecs. The control flow

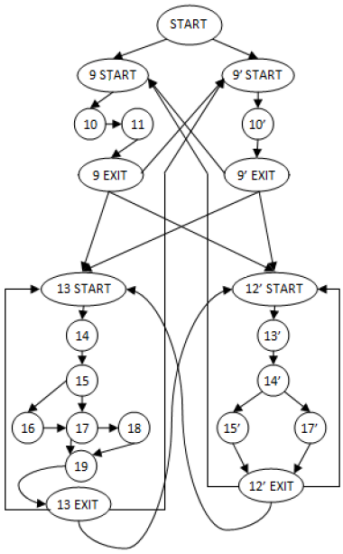

Fig. 6. RCFG of the sender/receiver model (from [96]). Line numbers refer to Fig. 2.

of the body of each message server is trivial but determining the control flow among the message servers themselves is difficult. The control flow of a Rebeca model is different from that of multi-threaded concurrent programs due to its reactive and event-based nature. In Rebeca, method calls are different from regular procedure calls or initiating a thread. A method call is performed by sending an asynchronous message; the body of the corresponding message server is guaranteed to execute later atomically. A method call does not transfer the execution control from the *caller* to the *callee* nor does it generate a new concurrent thread. The main issue here is to determine which message servers can potentially execute after a given message server has finished. This depends on the messages on top of the queues of all rebecs at that time. As an over-approximation, one can assume that after the execution of a message server is over, all message severs potentially have a chance for execution. For a more precise approximation, one can use the causality relations.

As Rebeca is a well-structured language, control dependence can be computed during the traversal of the abstract syntax tree. The above approximation implies that the last definition of each variable in a message server can reach the first statements of all message servers. However, there are no shared variables in Rebeca, therefore, the only manifestation of this approximation is that a change in the value of a state variable within a rebec, reaches the other message servers of the same rebec. Figure 6 shows the RCFG of the sender and receiver example.

The *Rebeca Dependence Graph (RDG)* captures the features of Rebeca as follows (see Figure 7): Each reactive class is modeled as a *class-object* node.

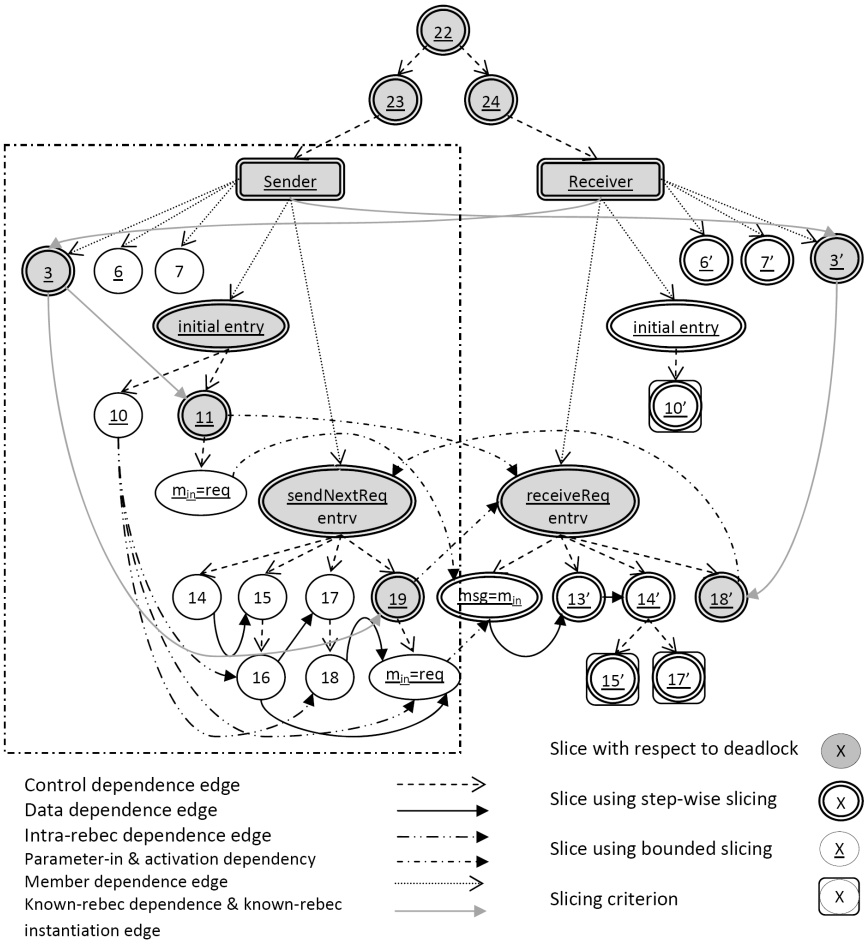

Fig. 7. RDG of the sender/receiver example (from [96]). Line numbers refer to Fig. 2.

Each message server is modeled by an *entry* node, a set of nodes representing its statements, and *data dependence* edges and *control dependence* edges modeling the existing dependencies within the body of the message server. The set of message servers of a reactive class are connected to the corresponding class-object node by *member dependence* edges. The member dependence edges ensure that a reactive class will be included in a slice if at least one of its message servers or state variables is included in that slice. Putting a message in a queue is represented by an *activation node*. In addition, an *activation edge* is used to connect the activation node to the *entry* node of the related message server and a *known-rebec dependence* edge is used to connect the activation node to the corresponding known rebec. Parameters of messages are modeled using *formal-in* and *actual-in* nodes along with *parameter-in* edges.

State variables are not shared among rebecs, but the message servers of each rebec share the state variables in the rebec. Therefore, there is a dependency between every message server using a variable and other message servers that assign a value to that variable. So, we introduce the notion of *intra-rebec dependency* to represent these kinds of dependencies. Considering the atomic execution of message servers, this dependency exists between the last statement of a message server assigning a value to a variable and the first use of that variable in other message servers (if the value of that variable is not changed in the body of the second message server before the first use).

To compute a slice from the resulting graph, four different algorithms are presented in [97, 96]. The first one is the traditional reachability algorithm which is used for static slicing, and the second algorithm is checking a model for deadlock. The third and forth techniques are based on iterative approximation and refinement of the model. In these techniques, an initial over-approximation of the original model is computed, and the model is subsequently refined based on the results of verification (alike in counter-example guided abstraction refinement - CEGAR [21]). The goal of these techniques is to find the minimal specification that satisfies a property, or otherwise a non-spurious counter-example. The reduced model is verified; if a spurious counter-example is found, then the model is refined to include more variables and the verification-refinement cycle is repeated. The *step-wise* slicing, starts with a reduced model including only the variables that construct the property, and the *bounded* slicing, is based on the nondeterministic assignments in Rebeca and user knowledge.

6.2 Related Work

A thorough general survey of slicing methods is provided in [70], and a survey of slicing techniques for object oriented programs is provided in [87]. In [34] slicing for concurrent object oriented programs is evaluated.

Compared to existing dependence graphs, the Rebeca dependence graph is simpler in several ways due to the asynchronous nature of communication, atomic execution of message servers, absence of shared data, and absence of procedure calls. In addition, Rebeca is an object-based language (as apposed to object-oriented), e.g., inheritance and polymorphism are not included in the language.

So, we do not need to deal with the complexities of dependence graphs designed for object-oriented languages.

We introduced a new type of dependency for message servers within a rebec, called intra-rebec dependency. This dependency cannot be captured as *inter-procedure dependency* because the sequence of execution of the message servers is not deterministic. It is also different from *interference dependency* because concurrency does not exist within a rebec. This dependency captures the concurrency in Rebeca, and unlike *interference dependence* in multi-threaded concurrency, it is transitive.

7 Schedulability Analysis for Timed Actors

Besides functional analysis of systems, it is also necessary to make sure that systems preserve a certain level of quality-of-service. In standard Rebeca, each rebec by default assumes a "First-Come, First-Served (FCFS)" strategy to run the messages in its queue. This is, however, not optimal when there are other measures like priorities or response time that play a role in the QoS. To optimize QoS, we enable rebecs to specify local scheduling strategies, e.g., based on fixed priorities, earliest deadline first, or a combination of such policies. Rebecs may require certain customized scheduling strategies in order to meet their QoS requirements.

In real-time modeling, a task specification indicates its execution time besides generation of other tasks; further, tasks have deadlines before which they must be scheduled and executed. Analyzing schedulability of a real time system consists of checking whether all tasks are accomplished within their deadlines. In Rebeca, message servers can be considered tasks and deadlines are associated to messages, as sending a message in this setting generates a new task.

We employed automata theory [53, 55] to provide a high-level framework for modular schedulability analysis of asynchronous reactive objects with local schedulers. In this framework, reactive objects are modeled abstractly using Timed Automata [8] so that analysis can be done in existing tools for example Uppaal [75]. At this level of abstraction, a method definition may abstract from the real computation and replace it by passage of time (as explained later in the example in Fig. 9).

In modular analysis, we analyze rebecs individually. To analyze a rebec in isolation, we need to restrict the possible ways in which its methods may be called; to this end, we make use of behavioral interfaces for rebecs. A behavioral interface specifies at a high level and in the most general terms how a rebec may be used. As in modular verification [72], which is based on assume-guarantee reasoning, individually schedulable rebecs can be used in systems *compatible* with their behavioral interfaces. Schedulability of such systems is then guaranteed. Compatibility being subject to state space explosion can be efficiently tested [55].

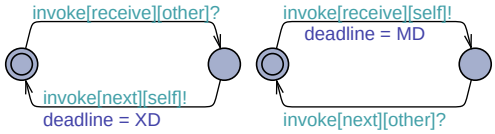

Fig. 8. Sender (left) and Receiver (right) Behavioral Interfaces

7.1 Real-time classes and rebecs in Timed Automata

Modeling behavioral interfaces. The abstract behavior of a rebec is specified in its behavioral interface. This interface consists of the messages the rebec may receive and send and provides an overview of the rebec behavior in a single automaton. A behavioral interface can also be seen as an abstraction (over-approximation) of the environments that can communicate with the rebec. A behavioral interface abstracts from specific method implementations, the queue in the rebec and the scheduling strategy.

In our example of sender-receiver, there are two interfaces (see Fig. 8). In this section, we use the shortened names 'receive' and 'next' instead of 'receiveReq' and 'sendNextReq', respectively. The Sender interface starts by outputting a 'receive' message; message communication is written in UPPAAL as 'invoke[m][r]' where 'm' denotes the message name and 'r' identifies the receiver. As dual to Sender, the Receiver interface inputs a 'receive' message at the beginning, which has a deadline 'MD'; to specify a deadline for a message we use a global variable which will be used by the scheduler automaton (described below after classes). Considering the symmetric model in Fig. 4, a reactiveclass may implement both of these interfaces.

To formally define a behavioral interface, we assume a finite global set $\mathcal{M}$ for method names. Sending and receiving messages are written as $m!$ and $m?$, respectively. A behavioral interface B providing a set of method names $M_B \subseteq \mathcal{M}$ is a deterministic timed automaton over alphabet Act^B such that Act^B is partitioned into two sets of actions:

- rebec outputs received by the environment: $Act^B_O = \{m?|m \in \mathcal{M} \wedge m \notin M_B\}$
- rebec inputs sent by the environment: $Act^B_I = \{m(d)!|m \in M_B \wedge d \in \mathbb{N}\}$

The integer d associated to input actions represents a deadline. A correct implementation of the rebec should be able to finish method m before d time units. The methods M_B must exist in the classes implementing the interface B. Other methods are sent by the rebec and should be handled by the environment.

Modeling classes. One can define a class as a set of methods implementing a specific behavioral interface. A class R implementing the behavioral interface B is a set $\{(m_1, A_1), \ldots, (m_n, A_n)\}$ of methods, where

- $M_R = \{m_1, \ldots, m_n\} \subseteq \mathcal{M}$ is a set of method names such that $M_B \subseteq M_R$;
- for all i, $1 \leq i \leq n$, A_i is a timed automaton representing method m_i with the alphabet $Act_i = \{m!|m \in M_R\} \cup \{m(d)! \mid m \in \mathcal{M} \wedge d \in \mathbb{N}\} \cup \{t?|t \in \mathcal{T}\}$;

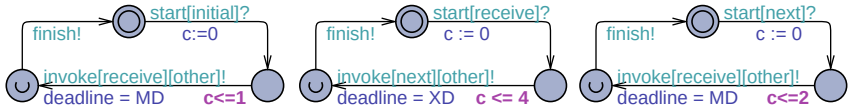

Fig. 9. Methods of a sender-receiver class

Classes have an *initial* method which is implicitly called upon initialization and is used for the system startup. Method automata only send messages or wait for replies while computations are abstracted into time delays. Receiving messages (and buffering them) is handled by the scheduler automata explained next. Sending a message $m \in M_R$ is called a self call. Self calls may or may not be assigned an explicit deadline. The self calls with no explicit deadline are called *delegation*. Delegation implies that the internal task (triggered by the self call) is in fact the continuation of the parent task; therefore, the delegated task inherits the (remaining) deadline of the task that triggers it. This mechanism is also handled by the scheduler.

Fig. 9 depicts the timed automata modeling the abstract behavior of the methods of a sender-receiver class. This class implements both the Sender and the Receiver interfaces in Fig. 8, therefore it needs to implement both 'receive' and 'next' methods (in addition to the 'initial' method). To enable starting and stopping method executions, all these method automata start by a synchronization on the 'start' channel and end by 'finish' channel. In between, in this example, each method only sends a message while the rest of the method is abstracted away into a time delay.

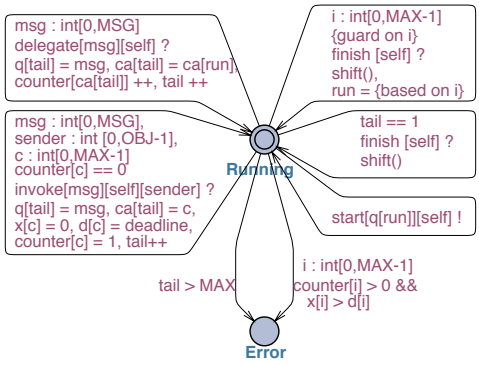

Fig. 10. A general scheduler automaton

Modeling schedulers. Fig. 10 shows the general structure of a scheduler automaton. The only thing not specified in this picture is the scheduling strategy. This automaton has the functionalities described below.

Queue. The queue is modeled using arrays in UPPAAL. For a message stored in q[i], the deadline is stored at d[ca[i]] and the clock x[ca[i]] keeps track of how long it has been in the queue. Delegation is modeled by reusing ca. The variable counter[i] holds the number of tasks that use clock x[i]. A clock is free if its counter is zero. When delegation is used, the counter becomes greater than one.

Input-enabledness. In this general scheduler automaton, there is an edge (left down in the picture) that allows receiving (at any time) a message on the invoke channel (from any sender). To allow any message and sender, select expressions are used. The expression msg : **int**[0, MSG] nondeterministically selects a value between 0 and MSG for msg. This is equivalent to adding a transition for each value of msg. Similarly, any sender (**sender** : **int**[0,OBJ−1]) can be selected. The selected message is put at the tail of the queue (q[tail] = msg), a free clock (counter[c] == 0) is assigned to it (ca[tail] = c) and reset (x[c] = 0), and the deadline value is copied (d[c] = deadline).

A similar transition accepts messages on the delegate channel. In this case, the clock already assigned to the currently running task (parent task) is assigned to the internal task (ca[tail] = ca[run]). In a delegated task, no sender is specified (it is always self).

Context-switch is performed in two steps (without letting time pass). When a method is finished (synchronizing on finish channel), it is taken out of the queue (by shift()). If it is not the last in the queue, the next method to be executed should be chosen based on a specific scheduling strategy (by assigning the right value to run). For a *concrete* scheduler, the guard and update of run should be well defined. If run is always assigned 0 during context switch, the automaton serves as a First Come First Served (FCFS) scheduler. An Earliest Deadline First (EDF) scheduler can be encoded using a guard like:

```
i < tail && i != run &&
forall (m : int[0,MAX-1])
( (m == run) ||
  (x[ca[i]] - x[ca[m]] >= d[ca[i]] - d[ca[m]])
)
```

and assigning run = (i < run) ? i : i−1 (because i is selected before shifting). The guard $x[a] - x[m] >= d[a] - d[m]$ makes sure that the remaining deadline of a, i.e., $x[a] - d[a]$, is bigger than or equal to the remaining deadline of m. The rest ensures that an empty queue cell (i < tail) or the currently finished method (run) is not selected.

If the currently running method is the last in the queue, nothing needs to be selected (i.e., if tail == 1 we only need to shift). The second step in context-switch is to start the method selected by run. Having defined start as an urgent channel, the next method is immediately scheduled (if queue is not empty).

Error. The scheduler automaton moves to the Error state if a queue overflow occurs (tail > MAX) or a deadline is missed (x[i] > d[i]). The guard counter[i] > 0 checks whether the corresponding clock is currently in use, i.e., assigned to a message in the queue.

7.2 Modular Schedulability Analysis

An rebec is an instance of a class together with a scheduler automaton. To analyze a rebec in isolation, we need to restrict the possible ways in which the methods of this rebec could be called. Therefore, we only consider the incoming method calls specified in its behavioral interface. Receiving a message from another rebec (i.e., an input action in the behavioral interface) creates a new task (for handling that message) and adds it to the queue. The behavioral interface doesn't capture (internal tasks triggered by) self calls. In order to analyze the schedulability of a rebec, one needs to consider both the internal tasks and the tasks triggered by the (behavioral interface, which abstractly models the acceptable) environment.

We can generate the possible behaviors of a rebec by making a network of timed automata consisting of its method automata, behavioral interface automaton B and a concrete scheduler automaton. The inputs of B written as $m!$ will match with inputs in the scheduler written as $m?$ and the outputs of B written as $m?$ will match outputs of method automata written as $m!$.

An rebec is schedulable, i.e., all tasks finish within their deadlines, if and only if the scheduler cannot reach the Error location with a queue length of $\lceil d_{max}/b_{min} \rceil$, where d_{max} is the longest deadline for any method called on any transition of the automata (method automata or the input actions of the behavioral interface) and b_{min} is the shortest termination time of any of the method automata [53]. We can calculate the best case runtime for timed automata as shown by Courcoubetis and Yannakakis [28].

Once a rebec is verified to be schedulable with respect to its behavioral interface, it can be used as an off-the-shelf component. To ensure the schedulability of a system composed of individually schedulable rebecs, we need to make sure their real use is *compatible* with their expected use specified in the behavioral interfaces. The product of the behavioral interfaces, called B, shows the acceptable sequences of messages that may be communicated between the rebecs. Compatibility is defined as the inclusion of the visible traces of the system in the traces of B [55].

To avoid state-space explosion, we test compatibility. A trace is taken from B and turned into a test case by adding **Fail**, **Pass** and **Inconc** locations. Deviations from the trace either lead to inconclusive verdict **Inconc** (meaning that no conclusions can be drawn from this test) when the step is allowed in B, or otherwise lead to **Fail** (meaning that a counter-example to compatibility is found). The submission of a test case consists of having it synchronize with the system. This makes the system take the steps specified in the original trace. The **Fail** location is reachable if and only if the system is incompatible with B along this trace. This testing method is sound and complete [55].

7.3 Related Work

Schedulability has usually been analyzed for a whole system running on a single processor, whether at modeling [38, 7] or programming level [27, 69]. We address

distributed systems where each rebec has a dedicated processor and scheduling policy. We propose a modular approach to schedulability analysis similar to the ideas of modular model checking [72]. The work in [40] is also applicable to distributed systems but is limited to rate monotonic analysis. Our analysis being based on automata can handle non-uniformly recurring tasks as in Task Automata [38]. In Task automata, however, a task is purely specified as computation times and therefore it cannot create sub-tasks.

RT-Synchronizers [93] are designed for declarative specification of timing constraints over groups of untimed actors. Therefore, they do not speak of schedulability of the actors themselves; in fact, a deadline associated to a message is for the time before it is executed and therefore cannot deal with the execution time of the task itself or sub-task generation.

In our approach, behavioral interfaces are key to modularity. A behavioral interface models the most general message arrival pattern for a rebec. In the literature, a model of the environment is usually the task generation scheme in a specific situation. However, a behavioral interface in our analysis covers all allowable scenarios of using the rebec, which in turn adds to the modularity of our approach; every use of the rebec foreseen in the interface is verified to be schedulable. Comparatively, for instance in TAXYS [27], this model of the environment can also be general enough to cover all uses of the program but it is used to analyze a complete program and is not used modularly.

In [12, 54], an extension to our approach is applied to accommodate explicit *release* statements and *replies* of the Creol language. Creol [61, 60] is a concurrent object-based language where the core of the language is similar to Rebeca and objects communicate only via asynchronous message passing. Asynchronous message passing in Creol is augmented with return values. Furthermore, Creol has explicit synchronization mechanisms, e.g., after sending a message, the caller may wait for a return value from the callee. A running method can decide to voluntarily *release* the control over processor, e.g., if the return values of a call are not yet available.

8 Extending Rebeca: Analyzing Self-Adaptive Models

Software systems are steadily becoming larger, more heterogeneous and long-lived. Flexible and scalable approaches are required for developing today's complex and evolving software-intensive systems. Hard-coded mechanisms make tuning and adapting long-run systems complicated. A prosperous practice is to enable such systems to continually evolve and adapt to situations not anticipated at development time.

In order to obtain adaptation, a major concern is *Flexibility*. Recently, the use of policies has been recognized as a powerful mechanism to achieve flexibility in adaptive and autonomous systems. With policies, one can *"dynamically"* specify the requirements in terms of high level goals. A policy is a rule describing the conditions under which a specified subject must, may or may not perform an action on a specific object.

Since self-adaptive systems are often complex and have a great degree of autonomy, it is more difficult to ensure that they behave as intended. Hence, it is of great practical importance to provide rigorous mechanisms for checking their correctness. To this end, *model-driven approaches* and *formal methods* can play a key role.

PobSAM (Policy-based Self-Adaptive Model) [67] is a flexible formal model to develop, specify and verify self-adaptive systems. It uses Rebeca as the language for specifying the functional behavior of systems. In order to build self-adaptive models, PobSAM adds two layers of *views* and *managers* on top of the actor layer. Analysis methods based on Rebeca model checking tools are proposed to check behavioral correctness, consistency of policies, and safety in the adaptation phase. Theories for checking behavioral equivalence and substitutability of two components are established. A mapping to Maude [82] is developed (not yet published) for more effective analysis of the models.

8.1 Motivating Example: Smart Home

As a motivating example, we describe a smart home based on the the example in [67]. In a home automation system, sensors are devices that provide a smart home with information about the physical properties of the environment. In addition, actuators are physical devices that can change the state of the world in response to these sensed data. The system processes the data gathered by the sensors, then it activates the actuators to alter the user environment according to the predefined set of policies. Smart homes can have different features, for example: (1) The lighting control can switch the lights on/off automatically, or adjust their intensity based on their placement in a room and according to the predefined policies. (2) Doors/Windows management controls windows and doors automatically. For instance, if windows have blinds, these should be rolled up and down automatically. (3) Heating control allows the inhabitants to adjust the house temperature to their preferred value. The heating control will adjust itself automatically in order to save energy.

The smart home system is required to adapt its behavior according to the changes in the environment. A typical system runs in normal, vacation and fire modes, and in each mode, it enforces different sets of policies to adapt to the current conditions. As some examples the policies defined for the lighting control module while the system runs in normal and fire modes can be as follows:

Defined policies in the normal mode

P1 Turn on the lights automatically when night begins.
P2 Whenever someone enters an empty room, the light setting must be set to default.
P3 When the room is reoccupied within T1 minutes after the last person has left the room, the last chosen light setting has to be reestablished.
P4 The system must turn the lights off, when the room is unoccupied.

Defined policies in the fire mode

P1 Turn on the emergency light.
P2 Disconnect power outlets.
P3 When the fire is extinguished, turn off the emergency light.

8.2 PobSAM Outline

A PobSAM model is the composition of three layers:

- The *actors layer* describes the functional behavior of the system and contains the computational entities. Rebeca is used to model these actors.
- The *managers layer* contains the autonomous managers. Managers are responsible for managing actors' behavior according to the predefined policies.
- The *view layer* is composed of a set of views that provide an abstraction of the actors' state for the managers. A view is a state variable, a function or a predicate applied to the state variables of actors.

The managers monitor the actors through views. Views provide managers with the required information about the actors. Each manager has a set of configurations containing adaptation policies and governing policies. The actors' behavior is directed by sending messages to them according to the governing policies. Adaptation policies are used for dynamic adaptation in response to the changing circumstances by switching between configurations.

In our example, actors are used at the functional level and model the sensors and actuators (e.g., the light actuator). The manager layer includes the policies in each configuration (e.g., the light controller would be a manager with the different policies for normal, fire and vacation configurations). The view layer acts as an abstract interface of the actors for the managers (e.g., a variable showing the intensity of each light and a variable showing the total intensity).

In this model, a new mode of operation, called the *adaptation mode*, is introduced to control the adaptation phase. Whenever an event which requires adaptation occurs, the relevant managers are informed. However, adaptation is not done immediately and the managers run in adaptation mode before switching to the next configuration. When the system reaches a safe state, the managers switch to the new configuration. This feature allows us to guide the adaptation process safely. There are two kinds of adaptation, called *loose adaptation* and *strict adaptation*. Under loose adaptation, the manager enforces old governing policies, whereas in strict adaptation, all events are ignored until the system passes the adaptation mode and reaches a safe state.

PobSAM has a formal foundation that employs an integration of algebraic formalisms and actor-based models. While the computational (functional) model of PobSAM is based on the actor-based semantics of Rebeca, the Configuration Algebra (CA) is proposed to specify the configurations of managers. A manager is formally defined as a tuple consisting of the set of possible configurations, the initial configuration, and the set of observable views for the manager. A

configuration is defined as a set of governing policies and a set of adaptation policies.

A governing policy consists of a priority, an event, a condition (a Boolean term) and an action. Events are generated when the execution of a message server is completed, when a message is sent, when a new actor is created, and when a specific condition in the system becomes true. The action part of a governing policy is specified using an algebraic theory in which the primitive action is sending a message to an actor (rebec). Action terms may be guarded; complex actions are constructed by sequential or parallel composition or by a nondeterministic choice among multiple actions. Whenever a manager receives an event, it identifies all the governing policies that are activated by that event. For each of the activated policies, if the policy condition evaluates to true, its action is triggered by sending a message to the relevant actors.

An adaptation policy is a prioritized rule that whenever triggered, drives the manager to the adaptation mode. The manager will switch to the new configuration after a safe state is reached. An adaptation policy consists of the priority of the policy, the triggering event, the condition of triggering the policy, the condition of applying the policy, the adaptation type (loose or strict), and the new configuration. Adaptation takes place in two phases. The adaptation policy implies that when the specified event occurs, and the triggering condition holds, if there is no other triggered adaptation policy with a higher priority, then the manager evolves to the strict or loose adaptation modes based on its type. When the condition of applying adaptation becomes true, the manager will perform adaptation and switch to the specified configuration.

8.3 Formal Analysis

We can perform different kinds of analysis on PobSAM models. In general, properties to be checked about an adaptive system can be categorized as adaptation properties, functional properties or a composition of both. Correctness properties of the functional layer (actors) of PobSAM models are application-specific. Correctness properties of the managers layer are related to the adaptation concerns (i.e., adaptation policies) or behavioral concerns (i.e., governing policies). Particularly, as policies direct the system behavior, it is required to understand and control the overall effect of governing policies on the system behavior. Governing policies often interact with each other and can cause undesirable effects. Hence, it is crucial to provide mechanisms to detect different kinds of policy conflicts. Furthermore, the correctness of the adaptation process of the PobSAM models, especially its stability, is an important property that needs to be verified.

In [68], we model PobSAM using Rebeca where actors and managers are modeled as rebecs, and views are modeled using global variables (as explained in Section 3.2 global variables are added to Rebeca for modeling system-level designs and can be used in a controlled way here, too). To enforce governing policies, a message server named enforce is considered for each manager rebec, which receives and handles events by interpreting the governing policies of the current configuration of the manager. While a manager is in normal or loose

adaptation mode, it handles events by enforcing the triggered governing policies based on the priority of the policies. Governing policies are expressed as a set of rules in the body of enforce. The conditional part of a governing policy is defined as a guarded expression. The policy context is defined in terms of the global state variables associated with the view layer. Moreover, a new generic classification of the conflicts that may exist among governing policies is introduced, and LTL patterns are proposed to express each type of these conflicts. A number of correctness properties of the adaptation process are also introduced. We use the model checking tools of Rebeca to detect policy conflicts and check the correctness of the adaptation phase. In addition to model checking, an approach based on static analysis of adaptation policies is presented to check system stability: if an adaptation by a manager leads to another adaptation, and this continues in a cycle then it causes an unstable state for the system which can be detected by a graph analysis technique.

Later, in [66], a behavioral equivalence theory is presented which helps in substitution of components and compositional reasoning. In dynamic environments such as the ubiquitous computing world, many systems must cope with variable resources, system faults and changing user priorities. In such environments, the system required to continue running with minimal human intervention, and the component assessment and integration process must be carried out automatically. Component assessment is identifying a component with the desired behavior that can replace another one. A possible solution to this problem relies on detecting the behavioral equivalence of components. Generally, we categorize behavioral equivalence of two components as context-independent or context-specific. Two components that are context-independent equivalent behave equivalently in any environment, whereas equivalence of two context-specific equivalent components depends on the environments in which they are running.

In [66], we present a context-independent behavioral equivalence theory to reason about managers, configurations, policies and policy actions. We develop semantic theories based on the notion of *splitting bisimulation* [11] and present sound and complete axiomatizations for this kind of bisimulation with respect to policy actions and governing policies. Furthermore, we introduce a new type of bisimilarity, called *prioritized splitting bisimulation*, to describe the behavioral equivalence of adaptation policies, configurations and managers. In [65], we develop an equational theory to analyze the context-specific behavioral equivalence of manager components based on a notion of behavioral equivalence, called state-based bisimulation. The view layer (i.e., the context) of the system is specified by a labeled state transition system. We extend our Configuration Algebra with new operators to consider the interaction of managers and the context and present the axioms of those operators. An important advantage of this equational theory is that it analyzes the behavioral equivalence of the manager layer using the view layer and independently from the actor layer.

8.4 Related Work

Dynamic adaptation is a very diverse area of research and different communities are concerned with this issue including autonomic computing, component-based systems, software architecture, coordination models, agent-based systems, etc. Structural adaptation has been given strong attention in the research community, and formal techniques have been extensively used to model and analyze dynamic structural adaptation (see [15]). *Structural adaptation* (or dynamic reconfiguration) is usually modeled using graph-based approaches (e.g. [112, 84]) or ADL-based approaches (e.g. [80, 90]).

Behavioral adaptation focuses on modifying the functionalities of the computational entities. Formal modeling and verification of adaptive systems at behavioral level is a young research area [18] and only a few research groups have already focused on this topic. As part of the RAPIDware project, Zhang et al. [123] proposed a model-driven approach for developing adaptive systems. In this approach, different contexts in which an adaptive program may run are specified by a formalism like temporal logic. The local properties of the program in each context are described formally. Then, a state-based model of the program in each context as well as the adaptation models for the adaptations of the program from one context to another are built. Different behavioral variants of a program are modeled as Petri Nets in [123]. Furthermore, they extend LTL with an "adapt" operator called A-LTL to specify adaptation requirements before, during and after adaptation [122] and introduce a model checking approach to verify the program formally. In another work [124], they propose a modular approach to verify adaptive programs.

Schneider et al. [100] present a method to describe adaptation behavior at an abstract level. After deriving transition systems from the system description, the system properties are verified using model checking techniques. In their later work [1], they propose a framework, MARS, for model-based development of adaptive embedded systems in which a model consists of a set of modules. A module may have different guarded configurations which are selected dependent on the current situation of the modules environment. The system is specified using Synchronous Adaptive Systems (SAS) [99] and is verified using theorem proving, model checking and specialized verification methods.

RAPIDware and MARS are on a different level of abstraction comparing to PobSAM. In these works, the system is described using a semantic-level state-based formalism while PobSAM uses high-level policies to control the system behavior and provide a high-level language to specify policies formally. Moreover, unlike PobSAM, configurations and the adaptation logic are fixed in RAPIDware and MARS. The ability to change configurations and the adaptation logic is vital to be able to model evolving adaptive systems. An act of adaptation in [122] results in a completely new program, but adaptation in PobSAM influences only the managers layer and the actors keeps running normally during adaptation. Thus, the adaptation semantics of PobSAM differs from that of A-LTL, however, both approaches consider safe adaptation.

A close area of research is coordination in which the interaction of objects can be controlled to achieve adaptation. While coordination models aim at decoupling interactions from computation and controlling interactions, PobSAM is concerned with controlling objects through controlling their behavior and decouples the *behavioral choices* and adaptation issues from the computational environment. ARC (Actor-Role-Coordinator)[94] and PAGODA (Policy And GOal based Distributed Architecture) [115, 116] are two actor-based coordination models in which meta-actors control interactions of actors. ARC controls objects interactions by manipulating message delivery, for instance via rerouting and reordering messages. In PAGODA, each coordinator is provided with a set of policies to coordinate actors where a simple policy may reorder messages, serialize requests and maintain a history of events.

9 Conclusion and Future Work

Several actor languages have been developed [91, 37, 93, 121, 119], and some of these languages are supported by model checking or testing tools [39, 101, 77]. In this paper, we focused on the imperative actor-based modeling language Rebeca, which has been designed in 2001 with the goal of providing a language for modeling concurrent and distributed systems with formal verification support. Throughout the paper, the language Rebeca and the supporting tools and techniques for analyzing Rebeca models are explained. Here, we will summarize our main design decisions in the language, its extensions and analysis techniques. We will also address some of the ongoing and future work. This list is far from complete.

The language and its extensions. The general design strategy of Rebeca has been to keep the language a pure actor-based modeling language with no synchronous communication. Rebeca is designed based on an operational view of the actor model introduced in [4, 5, 81]. The kernel of the Rebeca language is kept simple and only supports asynchronous non-blocking message passing. This has allowed us to provide powerful analysis methods based on specialized abstraction and reduction techniques. On the other hand, we extended Rebeca for a few specific domains, e.g., for hardware-software co-design (Section 3.2) [63, 92], and for globally asynchronous, locally synchronous systems (GALS) [108, 109]. To be used as a hardware-software co-design language, we added *global variables* in order to model events and signals in a system design, and *wait statements* to model the situations when a process in the system is waiting for a specific event. Our reduction techniques are extended to cover this extension of Rebeca. The same extension of Rebeca is used in designing self-adaptive policy-based systems (explained in Section 8). In the extension for modeling GALS, we added a formal notion of components to the language. Components interact only by asynchronous messages, while within each component, the reactive objects may communicate by synchronous messages. This offers a general framework which integrates, in a formally consistent manner, both synchrony and asynchrony.

Our formal verification approach is adjusted to reason about the open components. Certain properties are proven to be preserved when the model checked components are composed with other arbitrary components, and so, they can be plugged in a model relying on their behavior.

Analysis. Actor programming avoids the bugs inherent in shared-memory programming, but problems in incorrect sequential code within an actor, and problems in sequence of message passing still exist. These can cause deadlocks, race conditions, or bugs in the desired protocol. The biggest problem in analyzing actors is the growing number of sent messages (can be seen as events) that are not yet handled; this can quickly cause state-space explosion. In our tools, we allow the user to check the queue overflow condition and increase the size of the queue. In certain models the size of the queue is not bounded; in such situations, we need to have the option of running the system despite queue overflow. To handle an overflow, different policies can be taken, e.g., to overwrite the old messages or purge the new ones.

Rebeca has FIFO queues for the pending messages, which pose stronger ordering constraints in comparison to message bags used in many other actor languages. This preserves the happens-before relation [74, 77] while at the same time we consider all the possible interleavings for the execution of the rebecs. In our model, we have fewer message interleavings: for example between any pair of actors, the messages are processed in sending order, which is not the case for the models using bags. We consider atomic execution of methods, which is in line with the macro-step semantics of [5]. The combination of atomic execution and FIFO message queues causes even less message interleavings, for example, in the case where within a method we have more than one messages sent to the same rebec. These situations can be found by a simple static analysis of the code and if necessary be taken care of by a fine-grained execution of methods, or having a rebec in the middle that plays the role of a bag for the messages.

In analyzing Rebeca, we generally do not need to deal with the complications of programming languages, like complex data structures, or implementation details like managing the thread pools.

Future Work. The semantics and the established theories for Rebeca include dynamic creation of rebecs and dynamic topology, but the tools have to be extended to support these features of the language.

An ongoing work is a distributed implementation of the model checker. The BFS algorithm is especially suitable for parallel model checking [9]. This extension will distribute the state space across multiple computers, which will result in the ability to handle much bigger systems. In this work, we are investigating the applicability of call dependency graphs of Rebeca code which are similar to event diagrams of Clinger [26] but are derived using static analysis. Another approach to improve efficiency is using heuristics in model checking Rebeca. We are working on the application of best-first search algorithms using heuristics based on information from the message queues. Preliminary experimental results show the efficacy of the technique for some models [50].

A possible extension of Modere is to replace the rebec manager with a process/object manager for another language with a similar actor-based concurrency model. An ongoing work is integrating a Creol [62, 60] interpreter with Modere. Creol is based on concurrent objects (similar to actors). Creol has fine-grained interleaving and assumes no order on executing messages from the message bags. The semantics of Creol is implemented in the rewrite engine of Maude and as a result, execution and simulation of Creol models are currently possible. To the best of our knowledge, there exists however no efficient model checking tool for Creol. We expect that the state-space reduction techniques already developed in Modere could also be applicable to Creol (possibly with some adjustments) due to the similar concurrency model.

Acknowledgement

We wish to thank all the present and past members of the Rebeca group for their enthusiasm and hard work. In particular, we thank Hamideh Sabouri and Narges Khakpour for their help in writing the sections on slicing and self-adaptive models (respectively). Furthermore, we would like to thank Luca Aceto, Farhad Arbab and Mohammad Reza Mousavi for their comments on this paper. The work of the second author is supported by the EU FP7-231620 project called HATS.

References

1. Adler, R., Schaefer, I., Schüle, T., Vecchié, E.: From model-based design to formal verification of adaptive embedded systems. In: ICFEM. pp. 76–95 (2007)
2. Afra: a SystemC verifier, http://ece.ut.ac.ir/FML/afra.htm
3. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA (1990)
4. Agha, G., Mason, I., Smith, S., Talcott, C.L.: Towards a theory of actor composition. In: CONCUR: 3rd International Conference on Concurrency Theory. Lecture Notes in Computer Science, Springer-Verlag (1992)
5. Agha, G., Mason, I., Smith, S., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming 7, 1–72 (1997)
6. Agha, G.: The structure and semantics of actor languages. In: Proc. REX School/-Workshop on Foundations of Object-Oriented Languages. pp. 1–59. Springer-Verlag (1991)
7. Altisen, K., Gößler, G., Sifakis, J.: Scheduler modeling based on the controller synthesis paradigm. Real-Time Systems 23(1-2), 55–84 (2002)
8. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
9. Barnat, J., Cerná, I.: Distributed breadth-first search ltl model checking. Formal Methods in System Design 29(2), 117–134 (2006)
10. Behjati, R., Sabouri, H., Razavi, N., Sirjani, M.: An effective approach for model checking SystemC designs. In: Proceedings of IEEE ACSD'08 (2008)
11. Bergstra, J.A., Middelburg, C.A.: Preferential choice and coordination conditions. J. Log. Algebr. Program. 70(2), 172–200 (2007)

12. de Boer, F., Chothia, T., Jaghoori, M.M.: Modular schedulability analysis of concurrent objects in Creol. In: Proc. Fundamentals of Software Engineering (FSEN'09). vol. 5961, pp. 212–227 (2009)
13. Bosnacki, D., Dams, D., Holenderski, L.: Symmetric SPIN. International Journal on Software Tools for Technology Transfer (STTT) 4(1), 92–106 (2002)
14. Bosnacki, D., Donaldson, A.F., Leuschel, M., Massart, T.: Efficient approximate verification of promela models via symmetry markers. In: Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07). Lecture Notes in Computer Science, vol. 4762, pp. 300–315. Springer (2007)
15. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: WOSS. pp. 28–33 (2004)
16. Chang, P.H., Agha, G.: Supporting reconfigurable object distribution for customized Web applications. In: The 22nd Annual ACM Symposium on Applied Computing (SAC'07). pp. 1286–1292 (2007)
17. Chang, P.H., Agha, G.: Towards context-aware web applications. In: 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS). pp. 239–252 (2007)
18. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G.D.M., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software engineering for self-adaptive systems: A research roadmap. In: Software Engineering for Self-Adaptive Systems. pp. 1–26 (2009)
19. Cheong, E., Lee, E.A., Zhao, Y.: Viptos: a graphical development and simulation environment for tinyOS-based wireless sensor networks. In: Proceedings of the 3rd international conference on Embedded networked sensor systems, SenSys 2005. pp. 302–302 (2005)
20. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification (CAV 2002). Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer (2002)
21. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counter-example-guided abstraction refinement for symbolic model checking. JACM pp. 752–794 (2003)
22. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000)
23. Clarke, E.M.: The birth of model checking. In: Proceedings of the Symposium on "25 Years of Model Checking", Federated Logic Conference (FLOC'06) affiliated with CAV'06 (August 2006)
24. Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry reductions in model checking. LNCS, vol. 1427, pp. 147–158. Springer (1998)
25. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design 9(1/2), 77–104 (1996)
26. Clinger, W.D.: Foundations of actor semantics. Tech. rep., Cambridge, MA, USA (1981)
27. Closse, E., Poize, M., Pulou, J., Sifakis, J., Venter, P., Weil, D., Yovine, S.: TAXYS: A tool for the development and verification of real-time embedded systems. In: Proc. CAV'01. LNCS, vol. 2102, pp. 391–395. Springer (2001)

28. Courcoubetis, C., Yannakakis, M.: Minimum and maximum delay problems in real-time systems. Formal Methods in System Design 1(4), 385–415 (1992)
29. Donaldson, A.F., Miller, A.: Automatic symmetry detection for model checking using computational group theory. In: Proceedings of the International Symposium of Formal Methods Europe (FM'05). Lecture Notes in Computer Science, vol. 3582, pp. 481–496. Springer (2005)
30. Donaldson, A.F., Miller, A.: Extending symmetry reduction techniques to a realistic model of computation. Electronic Notes in Theoretical Computer Science 185, 63–76 (2007)
31. Donaldson, A.F., Miller, A., Calder, M.: Finding symmetry in models of concurrent systems by static channel diagram analysis. Electr. Notes Theor. Comput. Sci. 128(6), 161–177 (2005)
32. Donaldson, A.F., Miller, A., Calder, M.: Spin-to-Grape: A tool for analysing symmetry in Promela models. Electronic Notes in Theoretical Computer Science 139(1), 3–23 (2005)
33. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for dpll(t). In: Proc. Computer Aided Verification (CAV'06). LNCS, vol. 4144, pp. 81–94 (2006)
34. Dwyer, M.B., Hatcliff, J., Hoosier, M., Ranganath, V., Robby, Wallentine, T.: Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In: TACAS'06. LNCS, vol. 3920, pp. 73–89. Springer (2006)
35. Emerson, E.A.: Temporal and Modal Logic. In: J. van Leeuwen (ed.) Handbook of Theoretical Computer Science. vol. B, pp. 996–1072. Elsevier Science Publishers, Amsterdam (1990)
36. Emerson, E.A., Sistla, A.: Symmetry and model checking. Formal Methods in System Design 9(1–2), 105–131 (1996)
37. Erlang Programming Language Homepage, http://www.erlang.org
38. Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation 205(8), 1149–1172 (2007)
39. Fredlund, L.Å., Svensson, H.: Mcerlang: a model checker for a distributed functional programming language. SIGPLAN Not. 42(9), 125–136 (2007)
40. Garcia, J.J.G., Gutierrez, J.C.P., Harbour, M.G.: Schedulability analysis of distributed hard real-time systems with multiple-event synchronization. In: Proc. 12th Euromicro Conference on Real-Time Systems. pp. 15–24. IEEE (2000)
41. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Computer Aided Verification, 2nd International Workshop, CAV 90. pp. 176–185. Lecture Notes in Computer Science, Springer (1990)
42. Groote, J.F., Mathijssen, A., Reniers, M., Usenko, Y., Weerdenburg, M.V.: The formal specification language mCRL2. In: Proceedings of the Dagstuhl Seminar. MIT Press (2007)
43. Hendriks, M., Behrmann, G., Larsen, K.G., Niebert, P., Vaandrager, F.W.: Adding symmetry reduction to UPPAAL. In: Larsen, K.G., Niebert, P. (eds.) Proc. Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS '03, Marseille, France, September 6-7. LNCS, vol. 2791, pp. 46–59. Springer (2003)
44. 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 258, Department of Computer Science, MIT (Apr 1972)

45. Hewitt, C.: What is commitment? physical, organizational, and social (revised). In: Proceedings of Coordination, Organizations, Institutions, and Norms in Agent Systems II. pp. 293–307. Lecture Notes in Computer Science, Springer (2007)
46. Hewitt, C.: Orgs for scalable, robust, privacy-friendly client cloud computing. IEEE Internet Computing 12(5), 96–99 (2008)
47. Hewitt, C.: Actorscript(tm): Industrial strength integration of local and nonlocal concurrency for client-cloud computing. CoRR abs/0907.3330 (2009)
48. Hojjat, H., Sirjani, M., Mousavi, M., Groote, J.: Sarir: A Rebeca to mCRL2 translator (tool paper). In Proceedings of the 7th International Conference on Application of Concurrency to System Design (ACSD'07) (July 2007)
49. Holzmann, G.J.: The model checker SPIN. Software Engineering 23(5), 279–295 (1997)
50. IceRose homepage - projects, http://en.ru.is/icerose/applying-formal-methods/projects
51. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal methods in system design 9(1-2), 41–75 (1996)
52. Ip, C.N.: State Reduction Methods for Automatic Formal Verification. Ph.D. thesis, Department of Computer Science, Stanford University (1996)
53. Jaghoori, M.M., de Boer, F.S., Chotia, T., Sirjani, M.: Schedulability of asynchronous real-time concurrent objects. Journal of Logic and Algebraic Programming 78(5), 402–416 (2009)
54. Jaghoori, M.M., Chothia, T.: Timed automata semantics for analyzing Creol. In: Proc. Foundations of Coordination Languages and Software Architectures (FOCLASA'10). pp. 108–122. EPTCS 30 (2010)
55. Jaghoori, M.M., Longuet, D., de Boer, F., Chothia, T.: Schedulability and compatibility of real time asynchronous objects. In: Proceedings of Real Time Systems Symposium, Barcelona, 2008. pp. 70–79. IEEE (2008)
56. Jaghoori, M.M., Movaghar, A., Sirjani, M.: Modere: the model-checking engine of Rebeca. In: Haddad, H. (ed.) Proc. ACM Symposium on Applied Computing (SAC '06), Dijon, France, April 23-27. pp. 1810–1815. ACM (2006)
57. Jaghoori, M.M., Sirjani, M., Mousavi, M., Khamespanah, E., Movaghar, A.: Symmetry and partial order reduction techniques in model checking Rebeca. Acta Informatica 47, 33–66 (2010)
58. Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Movaghar, A.: Efficient symmetry reduction for an actor-based model. In: 2nd International Conference on Distributed Computing and Internet Technology. Lecture Notes in Computer Science, vol. 3816, pp. 494–507 (2005)
59. Jahania, M.: Using SAT-Solvers to model check Rebeca for data-centric applications - Master thesis, Sharif University of Technology (2008)
60. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (2007)
61. Johnsen, E.B., Owe, O., Arnestad, M.: Combining active and reactive behavior in concurrent objects. In: Langmyhr, D. (ed.) Proc. of the Norwegian Informatics Conference (NIK'03). pp. 193–204. Tapir Academic Publisher (Nov 2003)
62. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theoretical Computer Science 365(1-2), 23–66 (2006)
63. Kakoee, M.R., Shojaei, H., Ghasemzadeh, H., Sirjani, M., Navabi, Z.: A new approach for design and verification of transaction level models. In: ISCAS. pp. 3760–3763 (2007)

64. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: a comparative analysis. In: PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. pp. 11–20. ACM, New York, NY, USA (2009)
65. Khakpour, N., Jalili, S., Sirjani, M., Goltz, U.: Context specific behavioral equivalence of policy-based self-adaptive systems. In: ICFEM'11 (2011), to appear.
66. Khakpour, N., Jalili, S., Talcott, C.L., Sirjani, M., Mousavi, M.R.: Formal modeling of evolving adaptive systems. Submitted for publication.
67. Khakpour, N., Jalili, S., Talcott, C.L., Sirjani, M., Mousavi, M.R.: PobSAM: Policy-based managing of actors in self-adaptive systems. Electr. Notes Theor. Comput. Sci. 263, 129–143 (2010)
68. Khakpour, N., Khosravi, R., Sirjani, M., Jalili, S.: Formal analysis of policy-based self-adaptive systems. In: Proceedings of the 25nd Annual ACM Symposium on Applied Computing (SAC'10). pp. 2536–2543 (2010)
69. Kloukinas, C., Yovine, S.: Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems. In: Proc. ECRTS'03. pp. 287–294. IEEE CS (2003)
70. Krinke, J.: Advanced Slicing of Sequential and Concurrent Programs. Ph.D. thesis, Universitat Passau, Fakaltat fr Mathematic und Informatik (April 2003)
71. Krinke, J.: Context sensitive slicing of concurrent programs. ACM SIGSOFT Software Engineering Notes pp. 178–187 (2003)
72. Kupferman, O., Vardi, M.Y., Wolper, P.: Module checking. Information and Computation 164(2), 322–344 (2001)
73. Lamport, L.: Composition: A way to make proofs harder. In: Proceedings of COMPOS: International Symposium on Compositionality: The Significant Difference. Lecture Notes in Computer Science, vol. 1536, pp. 402–407. Springer-Verlag, Berlin, Germany (1997)
74. Lamport, L.: Ti clocks, and the ordering of events in a distributed system. Commun. ACM 21, 558–565 (July 1978)
75. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT 1(1-2), 134–152 (1997)
76. Lauterburg, S., Karmani, R., Marinov, D., Agha, G.: Evaluating ordering heuristics for dynamic partial-order reduction techniques. In: Rosenblum, D., Taentzer, G. (eds.) Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science, vol. 6013, pp. 308–322. Springer Berlin / Heidelberg (2010)
77. Lauterburg, S., Karmani, R.K., Marinov, D., Agha, G.: Basset: a tool for systematic testing of actor programs. In: SIGSOFT FSE. pp. 363–364 (2010)
78. Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits, Systems, and Computers 12(3), 231–260 (2003)
79. Leuschel, M., Massart, T.: Efficient approximate verification of B via symmetry markers. In: Proc. of the International Symmetry Conference, Edinburgh, UK. pp. 71–85 (2007)
80. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: In Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (1996)
81. Mason, I.A., Talcott, C.L.: Actor languages: Their syntax, semantics, translation, and equivalence. Theoretical Computer Science 220(2), 409–467 (1999)
82. Maude Homepage, http://maude.cs.uiuc.edu
83. McMillan, K.L.: A methodology for hardware verification using compositional model checking. Science of Computer Programming 37(1–3), 279–309 (May 2000)

84. Metayer, D.L.: Describing software architecture styles using graph grammars. Software Engineering, IEEE Transactions on 24(7), 521 –533 (1998)
85. Microsoft: Asynchronous agents library, http://msdn.microsoft.com/en-us/library/dd492627(VS.100).aspx
86. Miller, A., Donaldson, A.F., Calder, M.: Symmetry in temporal logic model checking. ACM Comput. Surv. 38(3) (2006)
87. Mohapatra, D., Mall, R., Kumar, .: An overview of slicing techniques for object-oriented programs. Informatica pp. 253–277 (2006)
88. NuSMV user manual, `http://nusmv.irst.itc.it/NuSMV/userman/index-v2.html`
89. Open SystemC Initiative: IEEE 1666: SystemC Language Reference Manual (2005), `www.systemc.org`
90. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE. pp. 177–186 (1998)
91. Ptolemy homepage, http://ptolemy.berkeley.edu/ptolemyII
92. Razavi, N., Behjati, R., Sabouri, H., Khamespanah, E., Shali, A., Sirjani, M.: Sysfier: Actor-based formal verification of systemc. ACM Trans. Embed. Comput. Syst. 10, 19:1–19:35 (2011)
93. Ren, S., Agha, G.: RTsynchronizer: language support for real-time specifications in distributed systems. ACM SIGPLAN Notices 30(11), 50–59 (Nov 1995)
94. Ren, S., Yu, Y., Chen, N., Marth, K., Poirot, P.E., Shen, L.: Actors, roles and coordinators - a coordination model for open distributed and embedded systems. In: COORDINATION. pp. 247–265 (2006)
95. de Roever, W.P., Langmaack, h., Pnueli, A. (eds.): Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 1997, Revised Lectures, Lecture Notes in Computer Science, vol. 1536. Springer-Verlag, Berlin, Germany (1998)
96. Sabouri, H., Sirjani, M.: Actor-based slicing techniques for efficient reduction of Rebeca models. Sci. Comput. Program. 75(10), 811–827 (2010)
97. Sabouri, H., Sirjani, M.: Slicing-based reductions for Rebeca. In: Electr. Notes Theor. Comput. Sci. vol. 260, pp. 209–224 (2010)
98. Scala Programming Language Homepage, http://www.scala-lang.org
99. Schaefer, I., Poetzsch-Heffter, A.: Using abstraction in modular verification of synchronous adaptive systems. In: Trustworthy Software (2006)
100. Schneider, K., Schuele, T., Trapp, M.: Verifying the adaptation behavior of embedded systems. In: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems. pp. 16–22. SEAMS '06, ACM, New York, NY, USA (2006)
101. Sen, K., Agha, G.: Cute and jcute: Concolic unit testing and explicit path model-checking tools. In: CAV. pp. 419–423 (2006)
102. Sirjani, M., Movaghar, A.: An actor-based model for formal modelling of reactive systems: Rebeca. Tech. Rep. CS-TR-80-01, Tehran, Iran (2001)
103. Sirjani, M., Movaghar, A., Iravanchi, H., Jaghoori, M., Shali, A.: Model checking Rebeca by SMV. In: Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'03). pp. 233–236. Southampton, UK (April 2003)
104. Sirjani, M., Movaghar, A., Mousavi, M.: Compositional verification of an object-based reactive system. In: Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'01). pp. 114–118. Oxford, UK (April 2001)
105. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.: Model checking, automated abstraction, and compositional verification of Rebeca models. Journal of Universal Computer Science 11(6), 1054–1082 (2005)

106. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.: Modeling and verification of reactive systems using Rebeca. Fundamenta Informatica 63(4), 385–410 (Dec 2004)
107. Sirjani, M., Shali, A., Jaghoori, M., Iravanchi, H., Movaghar, A.: A front-end tool for automated abstraction and modular verification of actor-based models. In: Proceedings of Fourth International Conference on Application of Concurrency to System Design (ACSD'04). pp. 145–148. IEEE Computer Society (2004)
108. Sirjani, M., de Boer, F.S., Movaghar, A.: Modular verification of a component-based actor language. Journal of Universal Computer Science 11(10), 1695–1717 (2005)
109. Sirjani, M., de Boer, F.S., Movaghar, A., Shali, A.: Extended Rebeca: A component-based actor language with synchronous message passing. In: Proceedings of Fifth International Conference on Application of Concurrency to System Design (ACSD'05). pp. 212–221. IEEE Computer Society (2005)
110. Sirjani, M., Movaghar, A., Iravanchi, H., Jaghoori, M.M., Shali, A.: Model checking in Rebeca. In: Arabnia, H.R., Mun, Y. (eds.) Proc. International Conference on Parallel and Distributed Processing Techniques and Applications. vol. 4, pp. 1819–1822. CSREA Press (2003)
111. Spin: Spin User Manual. http://spinroot.com/spin/Man/Manual.html
112. Taentzer, G., Goedicke, M., Meyer, T.: Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Theory and Application of Graph Transformations, Lecture Notes in Computer Science, vol. 1764, pp. 179–193. Springer Berlin / Heidelberg (2000)
113. Talcott, C.L.: Composable semantic models for actor theories. Higher-Order and Symbolic Computation 11(3), 281–343 (1998)
114. Talcott, C.L.: Actor theories in rewriting logic. Theoretical Computer Science 285(2), 441–485 (Aug 2002)
115. Talcott, C.L.: Coordination models based on a formal model of distributed object reflection. Electr. Notes Theor. Comput. Sci. 150, 143–157 (March 2006)
116. Talcott, C.L.: Policy-based coordination in PAGODA: A case study. Electr. Notes Theor. Comput. Sci. 181, 97–112 (2007)
117. Valmari, A.: A stubborn attack on state explosion. In: Computer Aided Verification, 2nd International Workshop, CAV 90. pp. 156–165. Lecture Notes in Computer Science, Springer (1990)
118. Vardi, M.Y.: Branching vs. linear time: Final showdown. In: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 1–22. TACAS 2001 (2001)
119. Varela, C., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. ACM SIGPLAN Notices 36(12), 20–34 (2001)
120. Weiser, M.: Program slicing. In Proceedings of the 5th international conference on Software engineering pp. 439–449 (1981)
121. Yonezawa, A.: ABCL: An Object-Oriented Concurrent System. Series in Computer Systems, MIT Press (1990)
122. Zhang, J., Cheng, B.H.C.: Specifying adaptation semantics. ACM SIGSOFT Software Engineering Notes 30(4), 1–7 (2005)
123. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Proceedings of the 28th international conference on Software engineering. pp. 371–380. ICSE '06, ACM, New York, NY, USA (2006)
124. Zhang, J., Goldsby, H., Cheng, B.H.C.: Modular verification of dynamically adaptive systems. In: AOSD. pp. 161–172 (2009)