Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 260 (2010) 209–224

www.elsevier.com/locate/entcs

Slicing-based Reductions for Rebeca

Hamideh Sabouri[a,1] Marjan Sirjani[a,b,2]

[a] *Department of Electrical and Computer Engineering*
University of Tehran
Tehran, Iran

[b] *School of Computer Science*
Institute for Studies in Theoretical Physics and Mathematics (IPM)
Tehran, Iran

Abstract

Slicing is a program analysis technique which can be used for reducing the size of the model and avoid state explosion in model checking. In this work a static slicing technique is proposed for reducing Rebeca models with respect to a property. For applying the slicing techniques, the Rebeca dependence graph (RDG) is introduced. As the static slicing usually produces large slices, two other slicing-based reduction techniques, step-wise slicing and bounded slicing, are proposed as simple novel ideas. Step-wise slicing first generates slices overapproximating the behavior of the original model and then refines it, and bounded slicing is based on the semantics of non-deterministic assignments in Rebeca. We also propose a static slicing algorithm for deadlock detection (in absence of any particular property). The applicability of these techniques is checked by applying them to several case studies which are included in this paper. Similar techniques can be applied on the other actor-based languages.

Keywords: Slicing, Actor-based languages, Rebeca, Model Checking, Verification, Reduction

1 Introduction

Model checking [4] is a formal verification technique for verifying concurrent systems against a number of specifications and can be used for developing more reliable systems. The main problem of model checking is the state space explosion problem and many techniques are developed to overcome this problem. These techniques include: abstract interpretation [5], data abstraction [9], predicate abstraction [12], slicing [31], partial order [23] and symmetry reductions [15].

To take advantage of model checking technique, one must first use a modeling language to represent the behavior of the system. Rebeca [27] (*Reactive Objects Language*) is an actor-based language with a formal foundation for modeling and verifying concurrent and distributed systems, which is designed in an effort to bridge

[1] Email: sabouri@ece.ut.ac.ir

[2] Email: msirjani@ut.ac.ir

doi:10.1016/j.entcs.2009.12.039

the gap between formal verification approaches and real applications. In [26] components are introduced for Rebeca language to encapsulate the tightly coupled reactive objects. This language is supported by a set of model checking tools [17,28,29].

Static slicing [31] extracts statements from a program which have a direct or indirect effect on a particular computation. One of the main approaches for slicing is using reachability analysis on program dependence graph.

For slicing Rebeca models a dependency graph should be constructed first. For this purpose we introduced a special dependency graph based on Rebeca semantics. This graph is less complicated than existing dependency graphs, due to the asynchronous nature of communication, atomic execution of message servers, absence of shared variables and absence of procedure calls (hence there is no need for *interference* or *summary* edges discussed in [19]). In addition, although Rebeca is an object-based language, we should not deal with complexities of dependence graphs designed for object-oriented languages, as features like inheritance and polymorphism are not included in the language. In the case of component-based models the corresponding subgraph of each component can be saved and reused when a component appears in another model.

For computing the slice from the resulted graph, four different algorithms are presented in this paper. The first one is the traditional reachability algorithm which is used for static slicing. The second algorithm is based on a simple novel idea and is used when we want to check a model against deadlock (unlike regular slicing algorithms there is no need to specify a property here). The idea is eliminating the statements that have no effect on any other statements.

In the third slicing algorithm, step-wise slicing, an overapproximation of the original model is computed and then based on the verification result, the reduced model is refined if needed. This algorithm starts by including the property variables in the model. Variables which have a direct effect on the value of the property variables, are also included in the model. These variables take a value using a non-deterministic assignment, in the reduced model. The other variables are eliminated from the model. Then, the reduced model is verified and if a spurious counter-example is found, the model is refined by including more variables in it.

The last algorithm, named bounded slicing, can be seen as an intermediate approach between static slicing and step-wise slicing. Static slicing preserves the property strongly but produces large slices including many variables. On the other hand, step-wise slicing only includes a few variables in the reduced model at the first step, but overapproximates the model and may require several refinement steps. In bounded slicing, the static slicing algorithm is bounded by non-deterministic assignments statements. The reason is that there is no statement in the program which could possibly affect the value of these assignments. User can bound the slicing process further by providing more variables to the bounded slicing algorithm. These are variables which their actual value is not important when checking a particular property, based on the user information. The bounded slicing algorithm replaces actual assignments which assign value to these variables with non-deterministic assignments and eliminates the other variables affecting the value of these variables.

Although the reduced model overapproximates the behavior of the original model, but the possibility of finding a spurious counter-example is reduced. The reason is that the variables are eliminated heuristically by the user (and not as an adhoc manner). However in the case of finding a spurious counter-example the model should be refined by adding more variables to it.

The contribution of this paper is to introduce slicing techniques for Rebeca.The available reduction techniques for Rebeca are symmetry reduction [18] and compositional verification [28,29]. The advantages of adding slicing techniques to the available reduction techniques for Rebeca are:

- **Combination with other reduction techniques**: Slicing can be used in combination with the other reduction techniques including compositional verification and symmetry reduction and make it possible to model check larger models.
- **Automatic processing**: The static slicing process is completely automatic and does not involve the user in the reduction process, comparing to the compositional verification approach in which the user should make a decision in selecting a number of components. Bounded slicing is applied automatically on a Rebeca model which uses non-deterministic assignment for assigning value to some variables. However the user can specify more variables (with non-deterministic values) for the bounded slicing algorithm to get a smaller slice. Step-wise slicing is not fully automatic in this work because the refinement process needs user interaction. But it can be improved to a fully automatic process and it is one of the future works.
- **Property preservation**: Static slicing is characterized by strong property preservation. This means that satisfaction and violation of a property in the original model can be directly concluded from the reduced model. In contrast, compositional verification overapproximates the model and the violation of the property in the reduced model does not necessarily implies the violation of the property in the original model. Both of the step-wise slicing and bounded slicing techniques overapproximate the model, however when bounded slicing is used, the possibility of finding spurious counter-examples is reduced.

The novelties in our technique can be summarized as:

- Introducing a special dependence graph for Rebeca due to the actor-base semantics of the language, which does not have the complexities of existing graphs. This graph can be applied for component-based Rebeca models, and in this case the subgraph of a component can be saved for further reuse.
- Presenting a slicing technique for slicing models to be verified against deadlocks (not a specific property).
- Presenting a slicing technique named step-wise slicing, which produces smaller slices by overapproximating the behavior of a Rebeca model.
- Presenting a slicing technique named bounded slicing, based on non-deterministic assignment to variables in Rebeca.

Same techniques (including the dependence graph and algorithms) can be applied to similar actor-based languages. In addition, these techniques can be used in combination with other reduction techniques.

This paper is structured as follows. The next section presents an overview of the related works. Section 3 briefly introduced the Rebeca language and program slicing technique. In Section 4 the Rebeca dependence graph is presented and in Section 5 different slicing algorithms are discussed. Section 6 explains the result of applying the slicing techniques to two case studies and the last section concludes the work.

2 Related Work

Static slicing has been used as a reduction technique in [1,6,21,13,2,24] for model checking purposes. In [7] an evaluation of using this technique for model reduction is presented. The result of [7] shows that slicing concurrent object-oriented source code provides significant reductions that are orthogonal to a number of other reduction techniques, and that slicing should always be applied due to its automation and low computational costs.

An approach named abstract slicing is presented in [14] which is based on abstract interpretation. Abstract slicing extends static slicing with predicates and constraints by using the program model as an abstract state graph, which is obtained by applying predicate abstraction to a program. For controlling the state space explosion problem, the abstract slicing is formulated in terms of symbolic model checking. In this abstraction technique, it can be determined under which conditions one statement might affect another. But for verification we may need to find out whether some condition might hold at all or not.

One of the ideas presented recently is incremental slicing [30]. It starts with a small, minimal part of the specification and successively adds further parts until either the property under interest holds on the slice or a real counterexample is found. This technique is applied to CSP-OZ [10]. The step-wise slicing technique presented in this paper uses the idea of overapproximating the behavior of the model and then refining it. However because of the different nature of the languages the way of applying the idea is different. In addition, in [30] the technique is applied to a simple automaton (comparing to our work in which the technique is applied to the dependency graph), therefore further comparison between these two techniques is not possible.

In [11,20] a technique is proposed for slicing synchronous reactive systems by introducing a new notion of slicing. In [11], this technique is applied to Argos language which is based on finite state machines. In [20] the Esterel language is considered which has a rich set of control constructs. The concentration of [20] is on modeling these constructs by defining new dependencies. The main difference of our work and this technique is the actor-based and asynchronous nature of Rebeca language.

```
reactiveclass Sender(2){
   knownrebecs{
      Receiver receiver;
   }
   statevars{
      int x;
      boolean y;
   }
   msgsrv initial(){
      x = 0;
      y = false;
      self.dataSend();
   }
   msgsrv dataSend(){
      if(y)
         x = x + 1;
      receiver.dataReceive(x);
      if(x == 5)
         x = 0;
      y = ?(true,false)
   }
}
```

```
reactiveclass Receiver(2){
   knownrebecs{
      Sender sender;
   }
   statevars {
      int a;
      boolean b;
   }
   msgsrv initial(){
      a = 0;
      b = false;

   }
   msgsrv dataReceive(int msg){
      a = msg;
      if(a == 5)
          b = true;
      else
         b = false;
      sender.dataSend();
   }
 }
```

```
main {
   Sender sender(receiver):();
   Receiver receiver(sender):();
}
```

Fig. 1. An example of a Rebeca model

3 Preliminaries

3.1 Rebeca

Rebeca [27] is an actor-based language for modeling concurrent and distributed systems as a set of *reactive objects* which communicate via asynchronous message passing. A Rebeca model consists of a set of *reactive classes*. Each *reactive class* contains a set of *state variables* and a set of *message servers* in which the body of the message servers is executed atomically. In a Rebeca model there is a set of *rebecs* (*reactive objects*) which are concurrently executed. *Rebecs* are encapsulated reactive objects, with no shared variables. Each *rebec* is instantiated from a *reactive class* and has a single thread of execution which is triggered by reading messages from an unbounded queue. Each message specifies a unique method to be invoked when the message is serviced. When a message is read from the queue, its method is invoked and the message is deleted from the queue. Each *rebec* has an *initial message server*, and in the initial state the queue of the *rebec* is empty and its statement to be executed is the first statement of the *initial message server*.

In [26], components encapsulate tightly coupled *reactive objects* which may have synchronous communication. The behavior of each component is like a *reactive object* and in the simplest case each *reactive object* is a component itself. In this paper we abstract from the internal synchronous communication as this is not natural behavior for actors.

Figure 1 is a very simple Rebeca example to show the syntax and semantics of Rebeca and our slicing techniques. This example is similar to alternating bit protocol, but we simplified it by putting a non-deterministic assignment instead of receiving a real acknowledgement by the sender.

In this example there exists a sender which sends a number of messages to a receiver. According to the non-deterministically chosen value of variable y, the sent message may be a new message or the previous message. After sending the last message this scenario starts over again. On the receiver side, after receiving the last message the value of a boolean variable named b is set to *true*. A possible property for this example is *G(F(b == true))* which checks whether the last message is finally received by the receiver. The property is an LTL (Linear Temporal Logic) formula in which G denotes *globally* and F denotes *Finally*.

3.2 Slicing

In general, slicing [31] is an analysis technique which is widely used in debugging, testing, maintenance and program comprehension. Program slicing, is first introduced as a decomposition technique that extracts statements relevant to a particular computation, from a program. A program slice consists of the parts of a program that potentially affect the values computed at some point of interest (referred to as a slicing criterion). In general, it is undecidable if a slice is minimal [31] and one of the attempts of slicing algorithms is to make the computed slice more precise.

The slicing technique has been improved to support concurrent and object oriented programs in addition to sequential programs. Each of these techniquees are described briefly in the following sections.

3.3 Slicing Sequential Programs

Slicing sequential programs can be divided into slicing programs without procedure and slicing programs with procedures.

For slicing programs without procedures, a reachability algorithm is performed on the program dependence graph (PDG) [16]. The PDG mainly consists of nodes which represent the statements of a program and two types of dependence edges: *Control dependence* edge that exists between two statement nodes if one node controls the execution of the other node. *Data dependence* edge that exists between two statement nodes if assigning value to a variable at one statement might reach the usage of the same variable at another statement.

In slicing programs with procedures, a two phase reachability algorithm is performed on the system dependence graph (SDG) [25]. The system dependence graph is a collection of procedure dependence graphs, one for each procedure. A procedure dependence graph contains nodes representing the procedure statements and *control* and *data dependence* edges. In addition, it contains an *entry* node representing entry to the procedure and a set of *formal-in* and *formal-out* nodes for modeling parameter passing. In each call site there is a *call* node and a set of *actual-in* and *actual-out* nodes. A *call* edge connects a procedure call site node to the entry node of the related procedure. *Parameter-in* edges and *parameter-out* edges connect the *formal-in* and *formal-out* nodes to the *actual-in* and *actual-out* nodes, respectively.

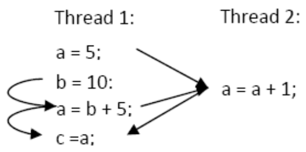


Fig. 2. An example of an imprecise slice in concurrent programs

3.4 Slicing Concurrent Programs

Slicing concurrent programs was first introduced in [3]. In [3] the notion of slicing is extended for concurrent programs and a graph-theoretical approach to slicing concurrent programs is presented. Slicing of concurrent programs had improved further in [22,19].

The SDG which is used for slicing sequential programs, is adopted to be used for slicing concurrent programs by adding a new dependence edge named *interference dependence*. In concurrent programs with shared variables, an *interference dependence* edge is added when a value is assigned to a variable in one thread and is used in another thread. The *interference dependence* edges are not transitive which may result an imprecise slice [19]. This problem can be solved by only considering realizable paths. However, even when only realizable paths are considered, the slice will not be as precise as possible. An example of this fact is shown in Figure 2 in which the *a=5* statement of *Thread 1* is included in a slice computed with respect to the *c=a* statement of this thread. But actually the *a=5* statement cannot affect the computation of the last statement.

4 Slicing Rebeca Models

4.1 Slicing Definition

When slicing is used in model checking for model reduction purposes, the definition of a slice slightly differs from the original definition which is used in software testing, debugging and maintenance. The reason is that in model checking the slicing is applied with respect to a property instead of a particular computation in a certain location of the program. Therefore the slice should be computed with respect to all of the points in which the involved variables in the property are taking a value.

4.2 Rebeca Dependence Graph

For slicing a Rebeca model, first the model should be transformed into an intermediate graph representation. After this step the slice can be computed through a graph reachability algorithm. The existing dependence graphs are not suitable for this purpose because they do not fulfill the requirements of the Rebeca language. In these graphs the emphasis is mainly on modeling procedures and procedure calls according to their context and in a further step concurrency feature is considered. In contrast, a Rebeca model does not include any procedure or procedure call and instead consists of an asynchronous communication through message passing.

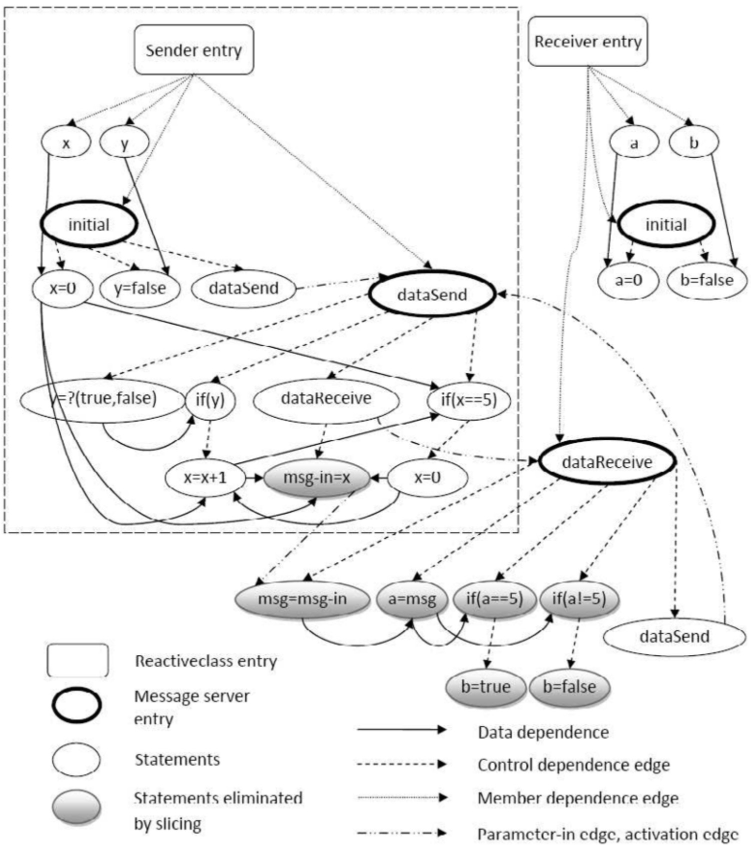


Fig. 3. RDG of the sender/receiver example

Therefore a special dependence graph for Rebeca named Rebeca dependence graph (RDG) is introduced. Here we discuss how RDG models Rebeca features:

(i) Reactive classes: an *entry* node is considered for each *reactive class*. The *member dependence* edges connect the *reactive class* entry node to each of its state variables and message servers.

(ii) Message servers: 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*.

(iii) Message passing: putting a message in a queue is represented through an *activation* node. In addition an *activation* edge is used for connecting the *activation node* to the *entry* node of the related *message server*. The parameters of the messages is modeled using *formal-in* and *actual-in* nodes as well as *parameter-in* edges.

(iv) Concurrency: as there is no shared variable between concurrent executing rebecs, there is no need for adding any special construct like *interference dependence* edge for this feature.

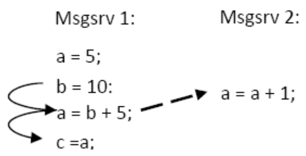


Fig. 4. Obtaining more precise slice with intra-rebec data dependence edge

(v) State variables: each *rebec* has its own set of state variables when executing concurrently with the other *rebecs*. But it should be considered that the *message servers* of the *rebec* are sharing these variables. Therefore some kind of dependency should exist between a *message server* using a variable and the other *message server* which is assigning value to that variable. This dependency is not a *data dependency* because it is not transitive and is not an *interference dependency* because concurrency does not exist within a *rebec* itself. We represent these kinds of dependencies with *intra-rebec data dependency*. According to atomic execution of the body of the *message servers*, this dependency exists between the last statement of a *message server* which is assigning value to a variable and the first use of that variable in another *message server* (if the value of that variable is not changed in the body of the second *message server* before the first use). In this way more precise slice can be obtained. Figure 4 shows how this idea solves the problem which is described in Figure 2 earlier.

(vi) main: as the important point in slicing Rebeca models is which state variables and message servers should be included in the slice, the *main* part of the model which just instantiates rebecs from the reactive classes, is not included in RDG.

For each component in the model, the related subgraph can be extracted from the RDG for further reuse. This can be done by selecting the *reactive classes entry* nodes related to the component and finding all of the nodes reachable from them. When finding reachable nodes, all of the edges are followed, except *activation* and *parameter-in* edges going to other *reactive classes*.

Figure 3 shows the RDG of the sender/receiver example which were discussed earlier. The dashed rectangle indicates a component including only the sender.

A dependency graph can be constructed for other actor-based languages in the same way:

- *Message servers* of the actors are similar to the *message servers* in Rebeca and can be modeled in a similar way.
- *Activation edges* can be used for modeling message passing between the actors.
- As there are no shared variables between the actors, there is also no need for *interference dependence edges*.
- The idea of *intra-rebec data dependency* can be used for modeling the variables of the actors.

```
S = { }                      //S denotes the slice of the model
 For each $c_i$               //$c_i$ (slicing criterion), w, v are nodes in RDG
    W = { $c_i$ }              //W denotes the working set
    S = S ⋃ { $c_i$ }
    While (W ≠ Ø)
        W = W/{w}            //removes one element
        For each v → w       // → denotes any kind of dependency
            If (v ∉ S)
                W = W ⋃ {v}
                S = S ⋃ {v}
```

Fig. 5. Static slicing algorithm

5 Slicing Techniques

In this section we describe slicing-based techniques which can be used for model reduction. As mentioned in the previous section, the *intra-rebec data dependency* edges are not transitive and precise slices can be computed by considering realizable paths. The purpose of the algorithms, presented in this section, is showing the main ideas of the slicing techniques, therefore for simplicity, we do not discuss the computation of realizable paths in these algorithms, in this paper.

5.1 Slicing Algorithm for RDG

After generating the RDG from the Rebeca model, the slice can be computed simply by a graph reachability algorithm. This algorithm should mark all of the nodes affecting the value of variables involved in the property. Figure 5 shows the slicing algorithm assuming that the value of involved variables is computed in nodes c_1,..,c_n and the final slice is stored in S.

5.2 Deadlocks

As we discussed earlier, a Rebeca model is sliced with respect to a property, but we may want to consider deadlock instead of any specific property. Here we present a simple idea for reducing Rebeca models using slicing-based techniques when checking the deadlock. For this purpose we search the RDG of the Rebeca model for the statement nodes which do not have an outgoing edge. In this way we are finding the statements which do not affect the other parts of the model. After finding these nodes we eliminate them with all of their incoming edges. This elimination may generate new nodes without any outgoing edge. Therefore the search operation should be repeated recursively until all the nodes have at least one outgoing edge. Figure 6 shows the algorithm of computing a slice for deadlock verification.

For example if we apply this technique to the sender/receiver example, a reduced RDG will be resulted. In Figure 3 the shadowed nodes are eliminated after applying this algorithm to the RDG.

5.3 Step-wise Slicing

Step-wise slicing technique generates a reduced model which overapproximates the original behavior of the model. Therefore we should use the counter-example guided

```
S = $\bigcup_i v_i$                              //at the beginning the slice includes all of the nodes
For each node $v_i$
    Check($v_i$)                                 //$v_i$ and w are nodes in RDG

Check($v_i$){
    If ($v_i \notin S$ or outgoing($v_i$) $\neq \emptyset$)   //outgoing($v_i$) denotes the set of outgoing edges of node $v_i$
        return
    If (outgoing($v_i$) = $\emptyset$)
        S = S/{$v_i$}                            //removes v from S
        For each w $\rightarrow$ $v_i$                    //$\rightarrow$ denotes any kind of dependency
            Check(w)
}
```

Fig. 6. Slicing algorithm for deadlocks

refinement technique [8].

The step-wise slicing process consists of a number of rounds and in each round the model is reduced, verified and refined (when a spurious counter-example is found). The process is terminated when the property is satisfied or by finding a feasible counter-example.

In this technique a set, named *selected variables* is considered and contains variables which should be included in the reduced model. In the first round this set only contains the variables involved in the property. The algorithm of computing a slice in each round is shown in Figure 7. In this algorithm statements that do not assign value to any variable (e.g. if statement) and assignments which assign value to a variable included in the *selected variables* set, are treated normally (i.e. similar to static slicing). But assignments which assign value to the other variables, are replaced by a non-deterministic assignment. In this case the *data dependence*, *intra rebec dependence* and *parameter-in* edges are not followed further by the algorithm.

After generating the slice, in the refinement step (if needed), user should choose at least one variable from the set of variables which were assigned by a non-deterministic assignment in that round. In the worst case all of the variables affecting the property will be included in the slice during the refinement steps. In this case, the result of this algorithm is equivalent to the result of static slicing.

5.4 Bounded Slicing

The main purpose of proposing this technique is the gap exists between the traditional static slicing method and step-wise slicing. The weakness of static slicing is that it usually generates a large slice and its advantage is that it preserves the property strongly. One the other hand step-wise slicing generates small slices (at least at the first stages of the algorithm) but it overapproximates the model. Additionally, it may take several rounds for getting a result, especially in large models.

In the bounded slicing technique we used the idea of non-deterministic assignments in Rebeca. However this technique can be applied to any other language supporting non-determinism. A non-deterministic assignment statement is not data dependent to any other statement so there is no *data dependence* edge, *intra-rebec* dependence edge or *parameter-in* edge that could be followed by the slicing algorithm. Thus, the algorithm is bounded by these assignments.

```
Set Selected;                          //Selected denotes the set of variables
                                       // which are included in the reduced model
For each $c_i$                         //$c_i$ (slicing criterion), v, w are nodes in RDG
    $W = \{ c_i \}$                    //W denotes the working set
    $S = S \bigcup \{ c_i \}$          //S denotes the slice
    While ($W \neq \emptyset$)
        $W = W/\{w\}$                  //removes one element
        For each $v \xrightarrow{C} w$ //$\xrightarrow{C}$ denotes control dependence and
                                       //activation edges
            If ($v \notin S$)
                $W = W \bigcup \{v\}$
                $S = S \bigcup \{v\}$

        If ($t \in$ Selected)          //t denotes the variable taking a value in node v
            For each $v \xrightarrow{D} w$   //$\xrightarrow{D}$ denotes data dependence, intra-rebec
                                       // dependence and parameter-in edges
                If ($v \notin S$)
                    $W = W \bigcup \{v\}$
                    $S = S \bigcup \{v\}$
        Else
            Non-deterministic assignment to t
```

Fig. 7. Step-wise slicing algorithm

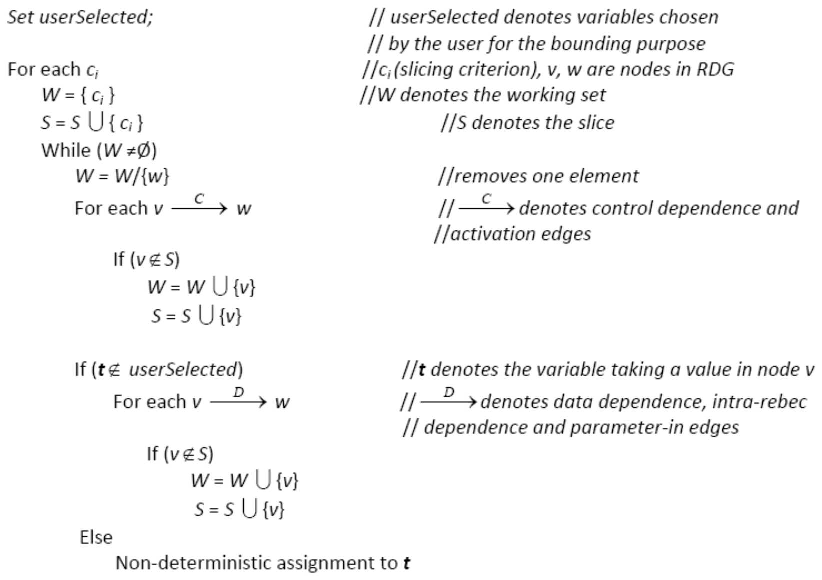
When a model contains non-deterministic assignments itself, the static slicing algorithm is bounded implicitly by these assignments. This can be considered an automatic version of this technique.

Additional non-deterministic statements can be added using user information. In this case the user provides the bounded slicing algorithm with a set of variables: *user selected* variables. These are variables which their value is not important when verifying the model against a specific property, based on user knowledge about the model. The algorithm in Figure 8 shows the bounded slicing algorithm. In this algorithm the assignments which assign value to variables included in the *user selected* set, are replaced by a non-deterministic assignment statement and *data dependence*, *intra-rebec dependence* and *parameter-in* edges are not followed further by the algorithm. In this way the slicing algorithm is bounded in certain points which are chosen by the user.

This technique overapproximates the model. However the possibility of facing spurious counter-examples using this approach is less than step-wise slicing because we tried to eliminate variables which have no effect on the property. In the case of finding a spurious counter-example the user can remove a number of variables from the *user selected* set and apply the algorithm again. The result of this algorithm is equivalent to the result of static slicing algorithm if the *user selected* variables set is empty.

6 Experimental Results

The proposed techniques were applied to a number of case studies. This section presents the results of reducing these case studies. The model checking is performed using Modere [17] on a computer with a 1.80 GHz CPU and 2038 MB of RAM.



```
Set userSelected;                      // userSelected denotes variables chosen
                                       // by the user for the bounding purpose
For each $c_i$                         //$c_i$ (slicing criterion), v, w are nodes in RDG
    $W = \{ c_i \}$                    //W denotes the working set
    $S = S \bigcup \{ c_i \}$                    //S denotes the slice
    While ($W \neq \emptyset$)
        $W = W/\{w\}$                            //removes one element
        For each $v \xrightarrow{C} w$           //$\xrightarrow{C}$ denotes control dependence and
                                                 //activation edges
            If ($v \notin S$)
                $W = W \bigcup \{v\}$
                $S = S \bigcup \{v\}$

        If ($\boldsymbol{t} \notin$ userSelected)          //t denotes the variable taking a value in node v
            For each $v \xrightarrow{D} w$           //$\xrightarrow{D}$ denotes data dependence, intra-rebec
                                                 // dependence and parameter-in edges
                If ($v \notin S$)
                    $W = W \bigcup \{v\}$
                    $S = S \bigcup \{v\}$
        Else
            Non-deterministic assignment to t
```

Fig. 8. Bounded slicing algorithm

The set of case studies includes:

- ***Commit problem (CP):*** There are n entities that are supposed to commit on performing an action. In the case that any of them disagrees, the action will be aborted.
- ***Dining philosophers problem (DP):*** A classic synchronization problem.
- ***Leader election problem (LE):*** A node should be selected as a leader in a ring of n nodes. It is supposed that each node knows the nodes next to it only. The leader is selected through the messages sent among the nodes.
- ***Sender receiver problem (SR):*** A sender and receiver communicate over a potential faulty communication line.
- ***CPU:*** The CPU is from http://www.es.ele.tue.nl/education/Computation/mmips-lab/ which contains the mmMIPS processor as well as several other variants of it.
- ***Pipeline (PL):*** A pipeline with four stage in which each stage performs a certain computation and passes the result to the next stage.
- ***Alarm clock (AC):*** A clock continually updates time and notifies clients registered for alarms.
- ***Sleeping barber (SB):*** A classic synchronization problem.
- ***Bounded retransmission protocol (BRP):*** A data link protocol used by Philips. The service it delivers is to transfer large files in a reliable manner, from a sender to a receiver.

Table 1 shows the result (number of states and time) of the model checking against the deadlock (shown by "DL" in the table) and properties (shown by "prop"

Table 1
Reduction gained for the case studies

Model	Complete Model	Static Slicing	Step-wise Slicing	Bounded Slicing
	# of states/time(s)	# of states/time(s)	# of states/time(s)	# of states/time(s)
CP (DL)	195745/12	**147327/9**	Not-applicable	Not-applicable
CP (prop)	195745/12	**147327/9**	**147327/9**	**147327/9**
DP (DL)	2864/3	2864/3	Not-applicable	Not-applicable
DP (prop)	122645/29	122645/29	122645/29	122645/29
LE (DL)	4627/1	4627/1	Not-applicable	Not-applicable
LE (prop)	9253/2	9253/2	9253/2	9253/2
SR (DL)	100026/2	100026/2	Not-applicable	Not-applicable
SR (prop)	250056/13	**48/1**	**48/1**	**48/1**
CPU (DL)	state-explosion	state-explosion	Not-applicable	Not-applicable
CPU (prop1)	state-explosion	**113404/17**	**110319/16**	**110319/16**
CPU (prop2)	state-explosion	state-explosion	**1809778/742**	**1809778/742**
PL (DL)	24772/1	24772/1	Not-applicable	Not-applicable
PL (prop1)	27022/2	**335/1**	**335/1**	**335/1**
PL (prop2)	24772/1	**346/1**	**346/1**	**346/1**
AC (DL)	state-explosion	state-explosion	Not-applicable	Not-applicable
AC (prop1)	state-explosion	**74/01**	**74/1**	**74/1**
AC (prop2)	state-explosion	**169588/4**	**169588/4**	**169588/4**
AC (prop3)	state-explosion	**6437/1**	**1928/1**	**1928/1**
SB (DL)	15762/1	**12978/1**	Not-applicable	Not-applicable
SB (prop1)	31316/6	**14420/2**	**14420/2**	**14420/2**
SB (prop2)	15762/1	**6322/1**	**6322/1**	**6322/1**
SB (prop3)	31629/6	**29722/4**	**29722/4**	**29722/4**
BRP (DL)	state-explosion	state-explosion	Not-applicable	Not-applicable
BRP (prop1)	state-explosion	state-explosion	**212599/6**	**212599/6**
BRP (prop2)	state-explosion	state-explosion	**3771783/390**	**3771783/390**
BRP (prop3)	state-explosion	state-explosion	**7762276/800**	**7762276/800**

in the table) for each case study. Bolded numbers indicate reductions in the state space. The static slicing technique reduces the number of states for most of the models. The *commit problem* and *sleeping barber* case studies show the applicability of the presented technique for slicing models against deadlocks and the *CPU* and *Bounded retransmission protocol* case studies show the advantage of step-wise slicing and bounded slicing over the static slicing technique: In smaller case studies the result of these techniques would be the same, but in larger examples in which the static slicing cannot avoid the state space explosion problem, step-wise slicing and bounded slicing techniques can help significantly.

7 Conclusion

In this paper we use slicing-based techniques for reducing the Rebeca models. A dependence graph named Rebeca dependence graph (RDG) is introduced for modeling the asynchronous nature of Rebeca. Three slicing-based techniques are used to compute the slices and each of them had a different reachability algorithm for computing the slice. In addition, a technique is proposed which reduces models that should be checked against deadlock.

Considering that the static slicing technique is automatic and the number of stages required in the step-wise slicing technique, it is recommended to apply these techniques in the following order: static slicing, bounded slicing, step-wise slicing.

In future work, we planned to find the main characteristics of the models which are best reduced by applying each of these techniques. Also, further investigation is ongoing to find more specializing techniques for Rebeca. Integrating these techniques with Rebeca verifier tool set is one of the other future works.

References

[1] Bozga, M., J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm and L. Mounier, *If: An intermediate representation and validation environment for timed asynchronous systems*, World Congress on Formal Methods (1999).

[2] Bruckner, I. and H. Wehrheim, *Slicing an integrated formal method for verification*, In ICFEM 2005: Seventh International Conference on Formal Engineering Methods, volume 3785 of LNCS (2005), pp. 360–374.

[3] Cheng, J., *Slicing concurrent programs–a graph-theoretical approach*, Proceedings of the First International Workshop on Automated and Algorithmic Debugging **749** (1993), pp. 223–240.

[4] Clarke, E. M., O. Grumberg and D. A. Peled, "Model Checking," The MIT Press, 2000.

[5] Cousot, P. and R. Cousot, *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, In Proceedings of POPL77 (1977), pp. 238–252.

[6] Dwyer, M. B., J. Hatcliff, M. Hoosier, V. Ranganath, Robby and T. Wallentine, *Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs*, TACAS (2006).

[7] Dwyer, M. B., J. Hatcliff, M. Hoosier, V. Ranganath, Robby and T. Wallentine, *Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs*, TACAS (H. Hermanns, J. Palsberg,Eds.), 3920, Springer (2006).

[8] E. M. Clarke, S. J. Y. L., O. Grumberg and H. Veith, *Counterexample-guided abstraction refinement for symbolic model checking*, JACM (2003), pp. 752–794.

[9] E.M. Clarke, O. G. and D. Long, *Model checking and abstraction*, In 19th ACM POPL (1992).

[10] Fischer, C., *CSP-OZ: A combination of Object-Z and CSP*, In H. Bowman and J. Derrick, editors, Formal Methods for Open Object-Based Distributed Systems(FMOODS 97) **2** (1997), pp. 423–438.

[11] Ganapathy, V. and S. Ramesh, *Slicing synchronous reactive programs*, Electronic Notes in Theoretical Computer (2002).

[12] Graf, S. and H. Saidi, *Construction of abstract state graphs with pvs*, In Proceedings of CAV97 (1997).

[13] Hatcliff, J., M. B. Dwyer and H. Zheng, *Slicing software for model construction*, Higher-Order and Symbolic Computation (2000), pp. 315–353.

[14] Hong, H., I. Lee and O. Sokolsky, *Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking*, SCAM, IEEE Computer Society (2005).

[15] Ip, C. and D. Dill, *Better verification through symmetry*, In International Conferenceon Computer Hardware Description Languages (1993).

[16] J. Ferrante, K. J. O. and J. D. Warren, *The program dependence graph and its use in optimization*, ACM Trans.Prog. Lang. Syst (1987), pp. 319–349.

[17] Jaghoori, M., A. Movaghar and M. Sirjani, *Modere: The model-checking engine of Rebeca*, ACM Symposium on Applied Computing - Software Verification Track (2006), pp. 1810–1815.

[18] Jaghoori, M. M., M. Sirjani, M. R. Mousavi and A. Movaghar, *Efficient symmetry reduction for an actor-based model*, ICDCIT **LNCS 3816** (2005), pp. 494–507.

[19] Krinke, J., *Context sensitive slicing of concurrent programs*, ACM SIGSOFT Software Engineering Notes (2003).

[20] Kulkarni, A. R. and S. Ramesh, *Static slicing of reactive programs*, Source Code Analysis and Manipulation, SCAM (2003).

[21] Millett, L. and T. Teitelbaum, *Issues in slicing promela and its applications to model checking, protocol understanding, and simulation*, Software Tools for Technology Transfer (2000), pp. 343–349.

[22] Nanda, M. and S. Ramesh, *Slicing concurrent programs*, Software Engineering Notes (2000), pp. 180–190.

[23] Peled, D., *All from one, one for all: On model checking using representatives*, In Proceedings 5th Workshop on Computer Aided Verification, number 697 (1993).

[24] Qi, X. and B. Xu, *An approach to slicing concurrent ada programs based on program reachability graphs*, IJCSNS International Journal of Computer Science and Network Security (2006).

[25] S. Horwitz, T. R. and D. Binkley, *Interprocedural slicing using dependence graphs*, ACM Transactions on Programming Languages and Systems (1990), pp. 26–61.

[26] Sirjani, M., F. de Boer and A. Movaghar, *Modular verification of a component-based actor language*, Journal of Universal Computer Science **11** (2005), pp. 1695–1717.

[27] Sirjani, M., A. Movaghar, A. Shali and F. de Boer, *Modeling and verification of reactive systems using Rebeca*, Fundamenta Informaticae **63** (2004), pp. 385–410.

[28] Sirjani, M., A. Movaghar, A. Shali and F. S. de Boer, *Model checking, automated abstraction, and compositional verification of Rebeca models*, J.UCS 11(6) (2005), pp. 1054–1082.

[29] Sirjani, M., A. Shali, M. Jaghoori, H. Iravanchi and A. Movaghar, *A front-end tool for automated abstraction and modular verification of actor-based models*, In Proceedings of ACSD **63** (2004), pp. 145–148. IEEE Computer Society.

[30] Wehrheim, H., *Incremental slicing*, ICFEM 2006 (2006), pp. 514–528.

[31] Weiser, M., *Program slicing*, In Proceedings of the 5th international conference on Software engineering (1981), pp. 439–449.