Electronic Communications of the EASST
Volume XXX (2015)

Proceedings of the 15th International Workshop on Automated Verification of Critical Systems (AVoCS 2015)

State Distribution Policy for Distributed Model Checking of Actor Models

Ehsan Khamespanah , Marjan Sirjani , MohammadReza Mousavi , Zeynab Sabahi Kaviani , and MohamadReza Razzazi

18 pages

Guest Editors: Gudmund Grov, Andrew Ireland
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

State Distribution Policy for Distributed Model Checking of Actor Models

Ehsan Khamespanah [1,2], Marjan Sirjani [2], MohammadReza Mousavi [3], Zeynab Sabahi Kaviani [1], and MohamadReza Razzazi [4]

[1] e.khamespanah@ut.ac.ir ,z.sabahi@ut.ac.ir
University of Tehran, School of Electrical and Computer Engineering
[2] ehsan13@ru.is, marjan@ru.is
Reykjavik University, School of Computer Science
[3] m.r.mousavi@hh.se
Halmstad University, Centre for Research on Embedded Systems
[4] Razzazi@aut.ac.ir
Amirkabir University of Technology, School of Computer Eng. and Information Tech.

Abstract: Model checking temporal properties is often reduced to finding accepting cycles in Büchi automata. A key ingredient for an effective distributed model checking technique is a distribution policy that does not split the potential accepting cycles of the corresponding automaton among several nodes. In this paper, we introduce a distribution policy to reduce the number of split cycles. This policy is based on the call dependency graph, obtained from the message passing skeleton of the model. We prove theoretical results about the correspondence between the cycles of call dependency graph and the cycles of the concrete state space and provide empirical data obtained from applying our distribution policy in state space generation and reachability analysis. We take Rebeca, an imperative interpretation of actors, as our modeling language and implement the introduced policy in its distributed state space generator. Our technique can be applied to other message-driven actor-based models where concurrent objects or services are units of concurrency.

Keywords: Distributed Model Checking, State Distribution Policy, Concurrent Objects, Actors, Rebeca

1 Introduction

Providing quality guarantees despite the ever-increasing complexity of computer systems has been and remains a grand challenge. Using formal methods, in general, and model checking [CES86] in particular, has been advocated as a response to this grand challenge. Model-checking tools explore the state space of the system exhaustively to make sure that a given property holds in all possible execution of a system. A major limiting factor in applying model checkers to practical systems is the huge amount of space and time required to store and explore the state space. Generating the state space of large-scale practical systems undoubtedly results in state spaces that cannot fit in the memory of a single computer.

Besides the traditional model-checking reduction techniques, distributed LTL model checking [GMS13, BHR13, VVFB11, BBC05, BBS01, BC06, BC0S06] is a well-known technique to

deal with huge state spaces. In distributed LTL model checking the state space is partitioned into some slices and each slice is assigned to a node. Theoretically, dividing cycle detection in a state space among a number of nodes increases the efficiency of model checking; however, unlike the sequential and the shared-memory parallel algorithms, the efficiency of these algorithms depends on the communication costs [OPE05]. The communication cost directly relates to the distribution policy of states among nodes, as detecting accepting cycles that span over many different nodes requires communication. Another, more fine-grained, representative of communication cost is the number of split transitions; a split transition is a transition between two states, where the hosts of source and destination states are different nodes.

In the present work, we tackle the state distribution policy problem in the state space generation of actor models [Hew72]. We introduce a new state distribution policy based on the the so-called Call Dependency Graph (CDG) of actor models. A CDG represents the abstract causality relation among messages of actors (Section 2). Our abstraction is akin to the dynamic representation of actor's event activation causality proposed by Clinger [Cli81].

The most primitive and widely used distribution policy is random state distribution [GMS13, BHR13, VVFB11, BC0S06]. Random state distribution policy distributes states among nodes based on their hash values. Random distribution policy guarantees load balancing. However, it is not an effective technique as cycles are scattered over many different nodes. In [BBC05], state distribution is performed based on the Büchi automata of the properties. LTL model checkers find accepting cycles in the synchronous product of the state space and the Büchi automata of LTL specifications. Therefore, distributing states based on the strongly connected components of the property Büchi automata avoids creation of split cycles in the state space. This way, there is no need for communication among nodes for detecting accepting cycles. In practice, the corresponding Büchi automata of LTL properties do not have many strongly connected components. Hence, this approach does not work efficiently in most practical cases.

In [OPE05], another state space distribution policy is suggested to improve the locality of cycles.This policy is based on the static analysis of an abstracted model and detects *may* or *must* transition relations among states [LT88]. Based on this analysis, if two states have a *must* relation, they should be stored in a same node. We use a similar idea in our state distribution policy and show that using the CDG improves the locality of cycles by reducing the split transitions in the state space. In other words, we find the *must* relations among the states of actor models using the CDG. Our technique is applicable to other service-oriented models where the unit of concurrency can be modeled as an autonomous active object and message passing is the only way of communication. To illustrate the applicability of our method, we implement it in the distributed model checker of Rebeca, which is an actor-based language for modeling and model checking of reactive systems (Section 3). The experimental results of using CDG illustrate that the number of split transitions is reduced significantly by up to 50% (Section 4). We also discuss possible extensions of our work and possible application domains for it (Section 5).

In a nutshell, the contributions of this paper are as follows:

- Introducing the notion of call dependency graph (CDG) for actor models as an abstract representation for message passing causality,
- Presenting the relation between the cycles in the CDG and the cycles in the state space of a model,

– Adapting the notion of CDG in order to define a state distribution policy,

– Implementing the proposed techniques in a distributed model checking tool, and

– Providing experimental results and measuring the efficiency of our technique by means of a number of case studies.

2 Call Dependency Graph of Actor Models

The actor model [Hew72, Agh90b, AMST97, Agh90a] is a well-established paradigm for modeling distributed and asynchronous systems. In this model, actors encapsulate the concept of concurrent behavior. Each actor provides services that can be requested by other actors through sending messages to the provider. Messages are put in the message buffer of the receiver; the receiver takes the message and executes the requested service, and consequently, may send some messages to other actors. The source code of a simple actor model is shown in Figure 1. This model consists of two actors $\texttt{ac}_1$ and $\texttt{ac}_2$, each of which provides two services. To start the execution of the model, some messages must be put in the message buffer of the actors (i.e., initially sent messages); this is specified in the `main` block (line 24). Sending a message is denoted by "actor_name.service_name()" (line 3).

We illustrate our approach using a Simple Actor Modeling language, called *SAM*, which contains the key features of the actor model. Below, we briefly introduce SAM, which is inspired by the earlier actor models, e.g., by Agha et al. in [AMST97] and by Sirjani et al. in [SMSB04].

Definition 1 (An Actor Model) An actor in SAM is a member of type $Actor = ID \times \mathscr{P}(Vars) \times \mathscr{P}(mtds)$, where $\mathscr{P}(\cdot)$ denotes power set and:

– *ID* is the set of actor identifiers,

– *Vars* is the set of variable names, and

```
1  Actor ac1 {
2     service msg1() {
3        ac1.msg2();
4        ac2.msg3();
5     }
6     service msg2() {
7        ac1.msg1();
8        ac2.msg4();
9     }
10 }
11 Actor ac2 {
12    int sv = 1;
13    service msg3() {
14       ac1.msg1();
15    }
16    service msg4() {
17       if (sv == 1)
18          sv = 4;
19       else
20          sv = 3;
21    }
22 }
23
24 main {
25    ac1.msg1();
26 }
```

Figure 1: An example of a simple actor model.

– *Mtds* is the set of method declarations.

In the above-given definition, the members of *Mtds* are tuples $(m, p, s) \in MName \times Vars^* \times Statement^*$, where m is the name of the message which must be served by this method, p is the lists of message parameters, and $s \in Statement^*$ is the sequence of statements compromising the method's body. The set of statements in SAM is limited to a number of preliminary statements defined below.

Definition 2 (SAM Statements) The set of SAM statements is defined as $Statement = Assignment \cup Condition \cup Send$ where:

– $Assignment = Vars \times Expr$ is the set of assignment statements. In Figure 1, we use $var = expr$ to denote the assignment statement $(var, expr)$.

– $Condition = BExpr \times Statement^* \times Statement^*$ is the set of conditional statements. In Figure 1, we use $if(bexpr)\ \sigma\ else\ \sigma'$ to denote the conditional statement $(bexpr, \sigma, \sigma')$.

– $Send = (ID \cup \{self\}) \times MName \times Expr^*$ is the set of send statements. In Figure 1, we use $a.m(e)$ to denote the send statement (a, m, e).

In the aforementioned items, *Expr* denotes the set of integer expressions defined using usual arithmetic operators (with no side effects). *BExpr* denotes the set of Boolean expressions defined using usual relational and logical operators. We dispense with further details of the syntax in this definition.

Based on Definition 1 and Definition 2, a SAM model is specified by $\mathscr{P}(Actor) \cup Send^*$ where the $Send^*$ term addresses the send statements of the `main` block (i.e. the initially sent messages). Note that since there may be more than one initial message for an actor, the send statements are ordered in a sequence not just a set of statements.

We define below the operational semantics of SAM in terms of a Labeled Transition System (LTS). In order to do this, the following assumptions and notations are required. We assume that the only possible communication mechanism among actors is asynchronous message passing. The type of messages is defined as $Msgs = ID \times MName \times ID \times (Vars \to Vals)$, where for a message $(a_1, m, a_2, arg) \in Msgs$, a_1 is the name of the sending actor, a_2 is the name of the receiving actor, m is the name of the message, and arg is a function for mapping argument names to their values. For the sake of simplicity and without loss of generality, we assume that the messages do not have arguments and are left out of the message signature in the remainder of this paper. The other assumption is that the received messages of an actor are stored in a FIFO mailbox. Hence, the mailbox of an actor is denoted by a sequence of messages, i.e., a member of $Msgs^*$.

Definition 3 (SAM Operational Semantics) For a given actor model *AC*, its labeled transition system $LTS(AC)$, is defined as a tuple $(S, s_0, Act, \to)$, where:

– S is the global state of a SAM model defined as a function $s : ID \to (Vars \to Vals) \times Msgs^*$, which maps an actors identifier to the local state of the actor, i.e., the values of its state variables and its mailbox content,

– $s_0 \in S$ is the initial state,

– $Act = Msgs$ is the set of action labels (sent messages).

– $\rightarrow \subseteq S \times Act \times S$ is the set of transitions, defined by the coarse-grained interleaving of actor message executions, by removing a message from their mailboxes, sending the messages in the body of the corresponding method and finally updating the global state (as the result of assignment statements). By coarse-grained interleaving, we mean that the sequence of messages in the body of a method are sent in an atomic sequence.

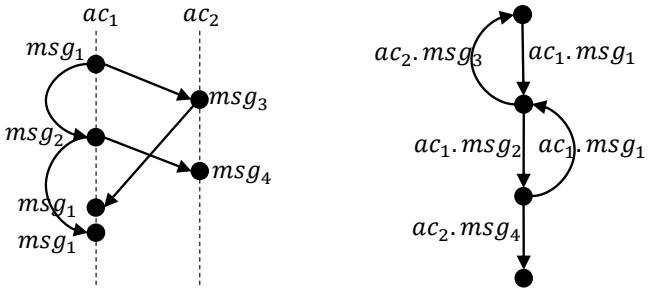
Using the operational semantics of actor models, Clinger's event diagram of actor models can be created. Clinger's event diagram comprise vertices (called *dots*) for each event, and edges (called *arrows*) that represent the activation relation of two events. Using dots and arrows, the runtime characteristics of an actor system is presented by the graph of activation relation of events. Clinger's event diagram is typically drawn using parallel vertical swim-lanes for actors, where the dots are placed respecting their sequential execution order. E Figure 2(a) presents the Clingers' event diagram of the example actor model of Figure 1. As shown in the actor model, message msg_1 is the first executing message (as it is put in the queue of $\texttt{ac}_1$ in the `main` block) which results in sending msg_2 and msg_3; hence, there is one *dot* with label msg_1, which is connected by *arrows* to two other *dots* with labels msg_2 and msg_3.

Clinger's event diagrams can be seen as detailed representations of CDG. Intuitively, a CDG represents the possible activation relations of events derived from a static analysis of the model. Hence, a CDG over-approximates the event activations in the Clinger's event diagram. The activation relation of events in a CDG can be extracted from the source codes of actor models. For example, as shown in Figure 2(a), the static activation relation between msg_1 and two messages msg_2 and msg_3 can be extracted from the source code of Figure 1. In addition, the execution of msg_3 results in the activation of msg_1, hence, the loop back to the topmost state of the CDG in Figure 2(b).

Definition 4 (Sent Messages) For a given message $msg = (a_1, m, a_2) \in Msgs$, the set of messages that can be possibly sent by a_2 to arbitrary actors as a result of serving m (which is in turn sent by a_1) is denoted by $snt(msg)$.
The set $snt(msg)$ is statically determined by an evaluation of the source code of an actor model. For a given message $msg = (a_1, m, a_2)$ assume that (m, p, s) is its corresponding method. The members of $snt(msg)$ is computed by the analysis of the statements of s, as depicted below, for a given SAM model $sam = acs \cup snt\_msgs$.

$$
\begin{aligned}
snt(msg) = \{ & msg' \in Msgs \mid \\
& msg = (a_1, m, a_2) \wedge msg' = (a_2, m', a_i) \wedge (a_2, vars, mtds) \in acs \rightarrow \\
& \exists stmts \in \mathscr{P}(Statements) \cdot (m, arg, stmts) \in mtds \wedge (a_i, m', \emptyset) \in stmts \\
& \}
\end{aligned}
$$

Definition 5 (Call Dependency Graph (CDG)) Given an actor model AC with $LTS(AC) = (S, s_0, Act, \rightarrow)$, its CDG is a finite labeled transition system $CDG(AC) = (V, v_0, Act, \hookrightarrow)$, where



(a) Clinger event diagram of an example actor model

(b) CDG of an example actor model

Figure 2: Clinger event diagram versus CDG of an example actor model.

$V \subseteq Act \times \mathscr{P}(Act)$ is the set of vertices (states), $v_0 \in V$ is the initial vertex (state), and $\hookrightarrow \subseteq V \times Act \times V$ is the set of edges (transitions). For two given states $u, v \in V$, there is $(u, act, v) \in \hookrightarrow$ if and only if $u = (act, pm)$ and $\exists (a_i, m', a_j) \in pm$ such that $v = ((a_i, m', a_j), snt(a_i, m', a_j))$. This way, the initial state of CDG is defined as $v_0 = (\varepsilon, pm)$ where $(\varepsilon, m, a_i) \in pm$ and there is a send statement in form of $a_i.m()$ in the `main` block of AC.

Next, we show that the abstract notion of the CDG reflects the cycles of the state space; more precisely, our goal is to show that each cycle in the LTS of an actor model can be projected into at least one cycle in the corresponding CDG.

Definition 6 (Labels, Sub-Traces, Sub-Cycles, and Cycles) Given an actor model AC a finite trace tr of $LTS(AC) = (S, s_0, Act, \rightarrow)$ is any finite word $m_0, m_1, \ldots m_n \in M^*$ such that there is a sequence $s_0, s_1, \ldots s_n, s_{n+1}$ of vertices in $LTS(AC)$, where s_0 is the initial state and $(s_i, m_i, s_{i+1}) \in \rightarrow$ for $i = 0, 1, \ldots, n$.

The set of cycles of $LTS(AC)$, denoted by $Cycles(LTS(AC))$, is the set of all traces that start and end with the same message. A sub-trace of a cycle which starts and ends with the same message is called a sub-cycle. The set of all edge labels of a given trace tr is denoted by $Labels(tr)$.

Theorem 1 (Mapping $LTS(AC)$ Cycles into CDG Cycles) *Each cycle in $LTS(AC)$ as the state space of the actor model AC, has a sub-cycle in $LTS(CDG)$.*

In order to prove the theorem, we first prove the following lemma, which establishes a link between individual messages appearing in the cycles of the state space and those appearing in the CDG. Here *appear* means that the message is the label of at least one of the transitions of the state space.

Lemma 1 *For each extended message m appearing in $LTS(AC)$ of actor model AC, m also appears in $CDG(AC)$.*

Proof. Assume that there exist messages which appear in the state space but never appear in the CDG. Pick one such message m that is reachable with the shortest trace from the initial state.

Assume that m is sent in the body of a service m'. Due to the minimality assumption for m, m' should appear in the CDG and by the definition of CDG, m should appear in the CDG subsequent to the edge labeled m' (i.e., m' is the parent of m), contradicting our original assumption. □

We also need the following definition.

Definition 7 (Parent and ancestors) Assume that $LTS(AC)$ and a trace $m_k \to \cdots \to m_t \to m_j \to m_i \in tr(LTS(AC))$ are given, m_j is called the *parent* of m_i, and is denoted by $P(m_i)$. In addition, all messages from m_k to m_t are called the *ancestors* of m_i. □

Proof. Consider a cycle $c_{LTS} \in Cycles(LTS(AC))$ and an arbitrary label $m \in Label(c_{LTS})$; we claim that for each trace $m \to \ldots \to m'$ in the traces of c_{LTS}, there exists a sub-trace $m \to \ldots \to m'$ in $CDG(AC)$. Once we prove this claim, the theorem follows by taking $m \to \ldots \to m = c_{LTS}$ as the antecedent of the claim (then, it follows from the claim that there should exist a sub-cycle of c_{LTS} in the CDG, which was to be shown).

To prove the claim, we use induction on the length of the trace $m \to \ldots \to m'$. The base case follows from Lemma 1. Assume that the claim holds for all traces of length n or less and consider a trace $m \to \ldots \to m'$ of length $n+1$. Let M be the set of parents of m' in all cycles of CDG (by Lemma 1, m' should appear in at least one cycle of the CDG). There exists some $m_i \in M$ such that $m \to \ldots \to m' = m \to \ldots \to m_i \to \ldots \to m'$. The trace $m \to \ldots \to m_i$ is of length n (or less) and hence the induction hypothesis applies and a sub-trace of it appears in $CDG(AC)$. Since m_i is a parent of m' in the $CDG(AC)$, an edge labeled m' follows after m_i. Therefore $m \to \ldots \to m_i \to m'$ is a trace of CDG, which was to be shown. □

3 Using CDG in the Distributed Model Checking Algorithms

In this section, we show how CDG can be exploited to improve the efficiency of state distribution policy in distributed model checking algorithms. Besides the traditional model checking algorithms, distributed model checking is proposed to deal with huge state spaces [BBC05, BBS01, BC06, BC0S06, BCKP01]. In distributed model checking, the state space is partitioned into slices and slices are distributed among multiple nodes for exploration. Dividing the exploration of a state space among nodes increases the analysis efficiency, but the performance gain heavily depends on the communication required among the nodes. Therefore, decreasing the number of split transitions (transitions between two states of which their hosts are different) reduces the required communication and hence the model checking cost. To reduce the number of split transitions, different states distribution policies are proposed [BBS01, GHS01]. To this aim, these policies use the static analysis of the source codes of models. Here, we show that how using CDG of actor models results in a better distribution of states in the distributed model checking of actor models. In the following, we show that how the CDG-based policy is implemented for distributed BFS-based model checking algorithm.

3.1 BFS Model Checking

The BFS exploration algorithm, creates and explores the state space in a level-by-level fashion and examines the back edges of the state space graph for cycle detection (explained below). In the first step of the BFS algorithm, the Cartesian product of the initial state of the state space and the property is stored, is marked as visited, and its level is set to zero. Then, for each level the successors of the states of that level are generated by applying the successor function to both the state space and the property automaton and their level is set by increasing the current level by one; when there are no unexplored states in the next level, the algorithm terminates.

This algorithm can be implemented using two queues to manage states of each level. The first queue stores the current level states (CLQ) and the second one stores the successors of the CLQ states. The latter queue is called the next level queue (NLQ). In each iteration, the unexplored states of the CLQ are dequeued and their unvisited successors are generated. When all states of the CLQ are dequeued, the content of the NLQ is moved to the CLQ and the algorithm continues until the NLQ is empty, i.e., all successors of the states in the CLQ have been visited. There is no need to examine all visited states, because only *back edges* may create cycles. Figure 1 shows a pseudo code of this algorithm. The backward search algorithm done by function *CYCLE-DETECTION* the same as the algorithm given in [BC06].

Algorithm 1: $BFS\_MODEL\_CHECKING(initState)$ traverses a given state space level by level.

Input: The initial state *initState*
Output: The state space of the system

```
1  CLQ ← {initState}
2  NLQ ← ∅
3  Visited ← ∅
4  while CLQ ≠ ∅ do
5      foreach state s ∈ CLQ do
6          foreach state s' ∈ PREDECESSORS(s) do
7              if s' ∉ Visited then
8                  Visited ← Visited ∪ {s'}
9                  NLQ ← NLQ ∪ {s'}
10             else
11                 CYCLE_DETECTION(s')
12     CLQ ← NLQ
13     NLQ ← ∅
```

3.2 Distributed BFS Model Checking Algorithm

A major difference between the centralized- and the distributed BFS model checking (BFS-MC) algorithm is in storing the next level states. In the centralized BFS-MC, all newly generated system states are stored in the NLQ but in the distributed BFS-MC, some of them should be sent

to other nodes of the cluster. In other words, each state has a host node. The host of a state is the node that is responsible for storing the state and generating its successors. Line 8 of Algorithm 2 shows host assignment based on the random distribution. After finding the host, if the newly generated state host is the same as its parent's and it has not yet been visited, then the state is stored in NLQ. In contrast, if the newly generated state's host is another node, the state is sent to it. Then, the host node receives the new state and checks if the state is visited before. Therefore, checking whether a state is visited or not can be done locally.

The other difference between the centralized- and the distributed BFS-MC is in the cluster nodes synchronization phase at the end of each iteration (line 17). In the synchronization phase, nodes that finish processing their CLQ wait for other nodes to finish their work. Hence, after synchronization, all nodes have processed their CLQ states and are ready to continue the search for the next level. If none of the nodes have any new state to explore, the value of `allFinished` is set to `true` in line 17 to terminate the model checking.

Algorithm 2: *DISTRIBUTED_BFS_MODEL_CHECKING*$(initState, id)$ traverses a given state space level by level.

Input: The initial state *initState* (which is *null* if this node is not the host of the initial state) and the node's *id*
Output: The state space of the system

```
1  CLQ ← {initState}
2  NLQ ← ∅
3  Visited ← ∅
4  allFinished ← false
5  while ¬allFinished do
6      foreach state s ∈ CLQ do
7          foreach state s' ∈ PREDECESSORS(s) do
8              hostId ← HASH_VALUE(s')
9              if id = hostId then
10                 if s' ∉ Visited then
11                     Visited ← Visited ∪ {s'}
12                     NLQ ← NLQ ∪ {s'}
13                 else
14                     CYCLE_DETECTION(s')
15             else
16                 send(s', hostId)
17     allFinished ← SYNCHRONIZE_ALL()
18     CLQ ← NLQ
19     NLQ ← ∅
```

3.3 States Distribution Policy based on CDG

In the new state distribution policy, we find the set of active cycles for each state. The active cycles of each state are found in the CDG of the model and are based on the messages which are executed before reaching this new state. Without loss of generality, we base the definition of our distribution policy on the simple cycles in a CDG (i.e., cycles with no repetition of vertices). When exploring the transitions of a state, we store the states that belong to a cycle of the CDG on the same cluster node, (i.e., states with the same active cycles).

Definition 8 (Active cycles of a state) Consider an actor model A and its CDG $CDG(A) = (V, v_0, Act, \hookrightarrow)$. For a given state $v \in V$, the set of active cycles of v is the subset of $Cycles(CDG(A))$ containing all cycles in which the label m_j appear, where m_j is a label of one of the outgoing edges of state v.

The implementation of the new distribution policy is given in Algorithm 3. As shown in the input section, the CDG of the model is generated before model checking and it is given as an input to the algorithm. For the sake of simplicity, only differences between Algorithms 2 and the new algorithm are shown in Algorithm 3 (the common parts are shown by $\cdots$). Namely, line 8 of Algorithm 2 is replaced with the CDG-based distribution policy of lines 5 to 8.

Algorithm 3: *DISTRIBUTED_BFS_MODEL_CHECKING*$(initState, id, CDG)$ traverses a given state space level by level.

Input: The initial state *initState* (which is *null* if this node is not the host of the initial state), the node's *id*, and *CDG* as the call dependency graph of the model
Output: The state space of the system

```
1  ...
2  while ¬allFinished do
3      foreach state s ∈ CLQ do
4          foreach state s' ∈ PREDECESSORS(s) do
5              activeCycles ← ∅
6              foreach message msg ∈ ENABLED_MESSAGE(s') do
7                  activeCycles ← activeCycles ∪ CDG_CYCLES(CDG, msg)
8              hostId ← CHOOSE_CYCLE(activeCycles)
9              ...
10     ...
```

4 Experimental Results

We implemented CDG-based distribution policy for the BFS-based distributed model checking engine of Rebeca, an actor-based language with a Java like syntax (A brief description about Rebeca and how the CDG of a Rebeca model is obtained is described in Appendix A). We studied

the impacts of using CDG in the state space generation and the analysis against reachability properties, using the current implementation of the Rebeca distributed model checking toolset. The test platform has been Ubuntu 9.10 on a cluster of 2.2GHz Pentium 4 Core2 Duo with 2GB of RAM storage for each cluster node. We chose the size of each cluster based on the number of simple cycles in the CDG of each case study.

Three different case studies are used to compare the execution time and the memory consumption among centralized model checking, distributed model checking with random distribution policy, and distributed model checking with distribution policy based on CDG. The examples are the asynchronous resource manager (from Figure 3), dining philosophers and train controller.

In the dining philosophers model, there are a number of philosophers sitting at a round table. Between each adjacent pair of philosophers, there is a chopstick. To model such a behavior, each philosopher can be in one of the following states: thinking, hungry, or eating. A philosopher thinks for a while, and then stops thinking and becomes hungry. When the philosopher becomes hungry, she cannot eat until he owns both of the chopsticks to her left and right. When the philosopher is done eating she puts down the chopsticks and begins thinking again.

In the train controller model there are a number of trains on each side of the bridge. Trains arrive non-deterministically and the controller has to manage them in such a way that only one train passes the bridge at a time, because there is one railway on the bridge. Each train announces its arrival to the controller and the controller lets the train enter the bridge, if there is no other train on the bridge. If the bridge is full then the arrived train is put in a queue. The waiting trains will be served respectively. Each train should faithfully declare its departure to the controller. The Rebeca code of each case study can be found at the Rebeca homepage [fml].

Each example represents different pattern of communication and synchronization: dining philosophers example shows a ring topology, train controller and asynchronous resource manager show a star topology. In the dining philosophers example, each actor sends requests and responses to its left and right neighbors. In the train controller, the bridge controller behaves like a binary semaphore, whereas in the resource manager, the central node behaves like a counting semaphore.

Asynchronous resource manager is model checked for deadlock freedom with 4 to 7 clients (5 to 8 rebecs). The dining philosophers example is model checked for deadlock freedom with 2 to 5 philosophers (4 to 10 rebecs). The train controller model is model checked for deadlock freedom with 2 to 8 trains (3 to 9 rebecs). Tables 1 and 2 show the results.

In the CDG-based distribution policy in comparison to the random distribution, there is the overhead of cycle membership check and instead we have fewer split transitions and less communication among cluster nodes for cycle detection. Our results show that time-wise the gain exceeds the overhead.

As shown in Table 1 in the large enough cases, the number of split edges in the CDG-based distribution policy is 50% to 70% of the random distribution policy. In addition, memory consumption is reduced, because storing the split transitions requires storing endpoints host ids of the edges. This improvement is about 10% for the asynchronous resource manager and 5% for the train controller.

Table 2 shows the gain in the execution time that is about 8% for the asynchronous resource manager and 13% for the train controller in their largest versions. For the dining philosophers model, although the split cycles for the CDG-based policy are 52% of the random-based policy,

Problem	Size	#Transitions	#Split Transitions		
			Random	CDG	improvement
Asynch. Resource Manager	**2 clients**	94	39	36	8%
	3 clients	818	540	432	20%
	4 clients	7,76K	5,83K	4516	23%
	5 clients	83,19K	66,52K	50,46K	25%
	6 clients	1,02M	850,74K	635,14K	26%
	7 clients	14,34M	12,30M	9,01M	27%
Dining Philosophers	**2 phils**	408	196	107	46%
	3 phils	10,30K	6,97K	4,81K	31%
	4 phils	206,00K	154,76K	75,86K	51%
	5 phils	3,78M	3,02M	1,60M	47%
Train Controller	**2 trains**	86	46	36	22%
	3 trains	620	433	296	32%
	4 trains	4,46K	3,35	2,29K	32%
	5 trains	33,30K	26,66K	18,17K	32%
	6 trains	265,89K	221,61K	148,32K	34%
	7 trains	2,30M	1,97M	1,30M	35%
	8 trains	21,83M	19,11M	12,40M	36%

Table 1: Split edges in the random and the CDG-based distribution policies.

the execution time remains the same. This is due to the many back edges and the big cycles in the state space. In this case the CDG-based policy reduces the split edges but the effect of the reduction is negligible comparing to the time spent for backward search in detecting accepting cycles.

We also measured load balancing of cluster nodes for the two distributed policies. Random distribution results in balanced load for each node. In our experimental results, we could see that in random distribution, the deviation from the best distribution starts from 9% in small models and reduces to less than 1% for the larger one. In the CDG distribution this deviation starts from 13% and reduces to 1.28% for larger models.

In general the experimental results show that our technique outperforms random distribution when the size of the model is large enough. The gain increases as the size of the model grows. Also in our approach, the load balancing of cluster nodes converges to the optimum point in larger examples.

5 Discussion, Conclusion and Future Work

In this paper we introduced the Call Dependency Graph (CDG) for the actor-based modeling language Rebeca. The CDG is generated by a static analysis of the model and is an abstract graph capturing the causality of message passing among actors. We designated and proved a relation between the cycles in the CDG and the cycles in the state space. We devised a distribution policy for the distributed model checker of *Rebeca* based on the CDG. The new distribution policy increases the efficiency of distributed model checking by increasing the locality of the accepting cycles. Our new policy is implemented as an extension of breadth first search distributed model checking. Experimental evidence supports that this new policy improves cycle locality, and decreases model checking time and memory in practice.

As future work, we plan to improve our algorithm by duplicating states to avoid split cycle

Problem	Size	#States	Time (sec)		
			Centralized	Random	CDG
Asynch. Resource Manager	**2 clients**	51	0	0	0
	3 clients	344	0	1	1
	4 clients	2,86K	0	4	4
	5 clients	28,78K	1	6	6
	6 clients	344,24K	157	24	21
	7 clients	4,71M	$>$ 6 hour	1846	1704
Dining Philosophers	**2 phils**	183	0	1	1
	3 phils	3,06K	2	8	8
	4 phils	46,01K	381	209	209
	5 phils	675,56K	$>$ 6 hour	4821	4818
Train Controller	**2 trains**	46	0	0	0
	3 trains	250	0	1	1
	4 trains	1,51K	0	2	2
	5 trains	10,19K	1	3	3
	6 trains	76,64K	9	4	4
	7 trains	641,74K	1147	26	24
	8 trains	5,96M	$>$ 6 hour	3192	2789

Table 2: Time consumption for centralized and distributed model checking with the random and the CDG-based distribution policies.

creation such that all cycles can be detected locally. This comes at the cost of more memory consumption, and we need to define a set of criteria to balance between the increase in the size of state space, due to duplicating states, and the decrease in the verification time, due to localizing cycles. Moreover, we look for property classes for which our distribution policy guarantees localized cycles. Finally, we would like to investigate the effect of incorporating CDG into other analysis and reduction techniques such as slicing.

Acknowledgments. We thank the anonymous reviewers of AVoCS 2015 for their useful comments. The work of M.R. Mousavi has been partially supported by the Swedish Research Council (Vetenskapsrådet) with award number 621-2014-5057 (Effective Model-Based Testing of Parallel Systems) and the Swedish Knowledge Foundation (Stiftelsen för Kunskaps- och Kompetensutveckling) in the context of the AUTO-CAAS project.

Bibliography

[Agh90a] G. Agha. The Structure and Semantics of Actor Languages. In Bakker et al. (eds.), *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May 28 - June 1, 1990, Proceedings*. Lecture Notes in Computer Science 489, pp. 1–59. Springer, 1990.

[Agh90b] G. A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990.

[AMST97] G. Agha, I. A. Mason, S. F. Smith, C. L. Talcott. A Foundation for Actor Computation. *J. Funct. Program.* 7(1):1–72, 1997.

[BBC05] J. Barnat, L. Brim, I. Cerná. Cluster-Based LTL Model Checking of Large Systems. In Boer et al. (eds.), *FMCO*. Lecture Notes in Computer Science 4111, pp. 259–279. Springer, 2005.

[BBS01] J. Barnat, L. Brim, J. Stríbrná. Distributed LTL Model-Checking in SPIN. In Dwyer (ed.), *SPIN*. Lecture Notes in Computer Science 2057, pp. 200–216. Springer, 2001.

[BC0S06] L. Brim, I. Cerná, P. M. 0002, J. Simsa. How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. *Electr. Notes Theor. Comput. Sci.* 135(2):3–18, 2006.

[BC06] J. Barnat, I. Cerná. Distributed breadth-first search LTL Model Checking. *Formal Methods in System Design* 29(2):117–134, 2006.

[BCKP01] L. Brim, I. Cerná, P. Krcál, R. Pelánek. Distributed LTL Model Checking Based on Negative Cycle Detection. In Hariharan et al. (eds.), *FSTTCS*. Lecture Notes in Computer Science 2245, pp. 96–107. Springer, 2001.

[BHR13] J. Barnat, J. Havlícek, P. Rockai. Distributed LTL Model Checking with Hash Compaction. *Electr. Notes Theor. Comput. Sci.* 296:79–93, 2013.

[CES86] E. M. Clarke, E. A. Emerson, A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transaction on Programming Languages and Systems* 8(2):244–263, 1986.

[Cli81] W. D. Clinger. Foundations of Actor Semantics. Technical report, Cambridge, MA, USA, 1981.

[fml] Rebeca Home Page - Distributed Model Checking Section. http://www.rebeca-lang.org/wiki/pmwiki.php/Tools/RebecaDistributedModelChecker.

[GHS01] O. Grumberg, T. Heyman, A. Schuster. Distributed Symbolic Model Checking for μ-Calculus. In Berry et al. (eds.), *CAV*. Lecture Notes in Computer Science 2102, pp. 350–362. Springer, 2001.

[GMS13] H. Garavel, R. Mateescu, W. Serwe. Large-scale Distributed Verification Using CADP: Beyond Clusters to Grids. *Electr. Notes Theor. Comput. Sci.* 296:145–161, 2013.

[Hew72] C. Hewitt. 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, 1972.

[JMS06] M. M. Jaghoori, A. Movaghar, M. Sirjani. Modere: The Model-Checking Engine of Rebeca. In Haddad (ed.), *SAC*. Pp. 1810–1815. ACM, 2006.

[LT88] K. G. Larsen, B. Thomsen. A Modal Process Logic. In *LICS*. Pp. 203–210. IEEE Computer Society, 1988.

[OPE05] S. Orzan, J. van de Pol, M. V. Espada. A State Space Distribution Policy Based on Abstract Interpretation. *Electr. Notes Theor. Comput. Sci.* 128(3):35–45, 2005.

[Sir06] M. Sirjani. Rebeca: Theory, Applications, and Tools. In Boer et al. (eds.), *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*. Lecture Notes in Computer Science 4709, pp. 102–126. Springer, 2006.

[SMSB04] M. Sirjani, A. Movaghar, A. Shali, F. S. de Boer. Modeling and Verification of Reactive Systems using Rebeca. *Fundamenta Informaticae* 63(4):385–410, 2004.

[VVFB11] S. Vijzelaar, K. Verstoep, W. Fokkink, H. E. Bal. Distributed MAP in the SpinJa Model Checker. In Barnat and Heljanko (eds.), *Proceedings 10th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2011, Snowbird, Utah, USA, July 14, 2011*. EPTCS 72, pp. 84–90. 2011.

A Rebeca

Rebeca is an incarnation of the actor model. It comes equipped with an on-the-fly explicit-state LTL model-checking engine called Modere [JMS06]. Rebeca has a Java-like syntax and an operational semantics [Sir06, SMSB04]. Each Rebeca model consists of a number of *reactive classes*, each describing the type of a number of *actors* (called *rebecs* in Rebeca). We describe Rebeca language constructs using a simple resource manager model (see Figure 3).

```
1  reactiveclass CentralNode(3) {
2     knownrebecs {Client c1, c2, c3;}
3     statevars {int max;}
4     msgsrv initial() {
5        max = 5;
6     }
7     msgsrv register(int cnt) {
8        max = max - cnt;
9        if(sender == c1)
10          c1.ack();
11       else if(sender == c2)
12          c2.ack();
13       else if(sender == c3)
14          c3.ack();
15    }
16    msgsrv return(int cnt) {
17       max = max + cnt;
18       if(sender == c1)
19          c1.start();
20       else if(sender == c2)
21          c2.start();
22       else if(sender == c3)
23          c3.start();
24    }
25 }
26
```

```
27 reactiveclass Client(2) {
28    knownrebecs {CentralNode cn;}
29    statevars {
30       byte id;
31       boolean asked;
32    }
33    msgsrv initial(byte id2) {
34       id = id2;
35       self.start();
36    }
37    msgsrv start() {
38       asked = true;
39       cn.register(id);
40    }
41    msgsrv ack() {
42       asked = false;
43       cn.return(id);
44    }
45 }
46
47 main {
48    CentralNode cn(c1, c2, c3):();
49    Client c1(cn):();
50    Client c2(cn):();
51    Client c3(cn):();
52 }
```

Figure 3: Rebeca model for an asynchronous resource manager.

In this model, there are two reactive classes `CentralNode` and `Client`. Each reactive class declares a set of *state variables*, whose valuations define the local state of the actors of that reactive class. Following the actor model, communication takes place by actors sending asynchronous messages to each other. Each actor has a set of *known rebecs* to which it can send messages. For example, an actor of type `CentralNode` knows all the actors of type `Client` (line 2), to which it can send messages (e.g., lines 10, 12, and 14). Reactive classes declare the messages to which they can respond. The way an actor responds to a message is specified in its corresponding *message server*. An actor can change the value of its state variables through an assignment statement (line 34), make decisions through a conditional statement (line 18), and communicate with other rebecs by sending a message (line 19). Since communication is asynchronous, each actor has a *message queue*, from which it takes the next incoming message.

```
1  reactiveclass CentralNode(3) {
2     knownrebecs{ Client c1, c2, c3; }
3     msgsrv initial() { }
4     msgsrv register() {
5        if(sender == c1)
6           c1.ack();
7        else if(sender == c2)
8           c2.ack();
9        else if(sender == c3)
10          c3.ack();
11    }
12    msgsrv return() {
13       if(sender == c1)
14          c1.start();
15       else if(sender == c2)
16          c2.start();
17       else if(sender == c3)
18          c3.start();
19    }
20 }
```

```
21 reactiveclass Client(2) {
22    knownrebecs {
23       CentralNode cn;
24    }
25    msgsrv initial() {
26       self.start();
27    }
28    msgsrv start() {
29       cn.register();
30    }
31    msgsrv ack() {
32       cn.return();
33    }
34 }
35 main {
36    CentralNode cn(c1, c2, c3):();
37    Client c1(cn):();
38    Client c2(cn):();
39    Client c3(cn):();
40 }
```

Figure 4: Rebeca model for an asynchronous resource manager.

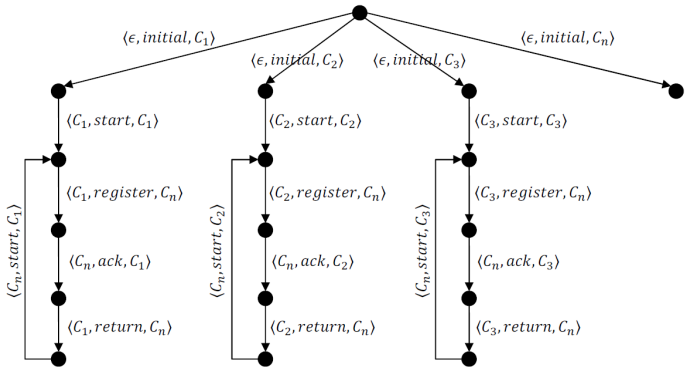
An actor takes the first message from its queue, executes the corresponding message server atomically, and then takes the next message (or waits for the next message to arrive).

For our resource manager, starvation-avoidance and resource-availability are two properties that are to be satisfied. Starvation-avoidance means that if a client asks for a resource, it will eventually receive it. Resource-availability property guarantees the existence of enough resources using the value of *max* state variable. The LTL formulas of these properties are given below.

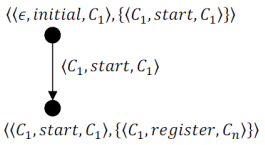
- Starvation-avoidance: $\mathbf{G}((c1.asked \rightarrow \mathbf{F}(\neg c1.asked)) \wedge (c2.asked \rightarrow \mathbf{F}(\neg c2.asked)) \wedge (c3.asked \rightarrow \mathbf{F}(\neg c3.asked)))$
- Resource-availability: $\mathbf{G}(cn.max > 0)$

A.1 Obtaining CDG of Rebeca Models

To obtain the CDG of a given Rebeca model, we first abstract away the original Rebeca model into a skeleton. This skeleton reflects the message communication structure of each reactive class together with the part of the control structure that is influenced by the signature of the message being processed. Then, we generate the CDG from the skeleton by applying Definition 5. The skeleton of the example of Figure 3 is depicted in Figure 4. The resulting CDG from the skeleton is depicted in Figure 5(a). The labels in the figure are the edge labels and the vertices are not labeled in order not to clutter the figure. A sample of vertex label, for the end points of the edge $\langle C_1, start, C_1 \rangle$ are shown in Figure 5(b).



(a) The call dependency graph of Figure 4.



(b) Labels of two states of Figure 5(a) which are connected by the edge $\langle C_1, start, C_1 \rangle$.

Figure 5: An example CDG which is extracted from the Rebeca model of Figure 3.