Runtime verification of distributed algorithms using high-level Petri nets

Fateh Latreche[1,*], Hichem Talbi[2]

[1]LIRE Laboratory, Abdelhamid Mehri Constantine 2 University, Algeria
[2]MISC Laboratory, Abdelhamid Mehri Constantine 2 University, Algeria

Abstract
Distributed algorithms play an important role for current IT applications and systems. The formal verification of the correctness of distributed algorithms is a challenging task. Distributed algorithms are frequently deployed on dynamic and scalable distributed systems. Therefore, these algorithms will generate a huge space-state and require an long checking time. The aim of this work is explore the use of high-level algebraic Petri nets, Recursive ECATNets, for the specification and formal verification of distributed algorithms. The proposed modelling formalism supports the inherent multi-threading aspect of distributed algorithms. Furthermore, it allows specifying compactly distributed algorithms with complex synchronisation constraints and advanced data structures. In order to show the effectiveness of our approach, the proposed model is applied on the *Zusuki and Kasami* mutual exclusion algorithm. Recursive ECATNets' models of distributed algorithms are implemented in this work using the Maude system. In order to alleviate the state explosion problem, the runtime dynamic verification approach is used to check whether a distributed algorithm respect the recursive ECTANet model.

Keywords
Distributed algorithms verification, Recursive ECATNets, Runtime verification, Maude system

1. Introduction

Thanks to the technological advances in the fields of computers and communication networks, distributed systems have become feasible. Distributed systems can be found at several facets of our daily life. This is due to the widespread use of wireless networks, mobile devices and Internet of Things. Distributed systems are now more and more able to meet the growing and renewable needs of users in terms of computing power and storage space.

Several definitions of the concept "distributed systems" exist. In [1], a distributed system is defined as a collection of interacting autonomous computing elements. The collection appears to their users as a single coherent system. This first definition reveals that each component of a distributed system may behave independently of other components. Despite this, the clients of the distributed system believe that they are dealing with a single coherent system, i.e. the existence of stand-alone components is hidden from users.

Leslie Lamport [2], defines a distributed system by : "A distributed system is one in which the

TACC 2023: Tunisian-Algerian Joint Conference on Applied Computing, November 06–08, 2023, Sousse, Tunisia
*Corresponding author.
fateh.latreche@univ-constantine2.dz (F. Latreche); hichem.talbi@univ-constantine2.dz (H. Talbi)
0000-0003-1220-6586 (F. Latreche); 0000-0001-7320-5312 (H. Talbi)

failure of a computer you didn't even know existed can render your own computer unusable". This definition suggests that a user is aware of the use of a distributed system, if the failure of another computer of the system prevents him/her from completing his/her work. The definition refers to the fault tolerance challenge. Due to the heterogeneity and the physical distribution of its components, parts of a distributed system may fail. In this case, the distributed system should automatically recover from failures without affecting the overall performance.

Distributed systems provide several advantages. The well-known advantage of a distributed system is that it enables resource sharing among users. This will facilitate collaboration and reduce costs. In addition, distributed systems improve calculation speed and efficiency thanks to the parallel execution of requests over multiple components [1].

Distributed computing has grown rapidly in recent years and has become able to tackle advanced topics. However, distributed applications can be safety-critical for users since an unwanted behaviour can lead to serious risks. Thus, the designers of distributed applications should ensure the accuracy of used algorithms before deploying them for real exploitation.

Checking the correctness of distributed algorithms was the aim of several studies. Formal-based methods are good candidates for specifying and checking distributed algorithms. They provide notation with well-defined semantics and are amenable to sound analyses.

The formal checking of distributed algorithms is challenging. This is due to multiple factors, such as the potential interleaving of events, the asynchronous communication and the possible failures [3]. Checking the correctness of distributed algorithms can be approached using model checking techniques and theorem proving. However, these techniques suffer from the state space explosion problem and the need for user intervention.

Some research works have proposed to combine model checking and theorem proving to model and check correctness of distributed algorithms. However, they have used restrictive specification notations. In addition, these works still need intervention from users with an appropriate level of experience.

In this paper, we propose to formally specify distributed algorithms using recursive ECATNets [4] and then check their correctness using a runtime verification approach. Recursive ECATNets have a graphical notation that facilitates design of distributed algorithms models. They also allow modelling the multithreading aspect inherent to distributed algorithms. The runtime checking approach is formal and provides a lightweight yet rigorous method for analysing the runtime behaviour of systems [5].

The paper is organised as follows: Section 2 presents some important related works. The third section gives some preliminary concepts. In section 4, we describe the proposed approach for modelling and checking at runtime distributed algorithms. Finally, section 5 gives some final remarks and future extensions.

2. Literature review

Ensuring the correctness of distributed algorithms was the aim of several research works [6]. These research efforts have focused on proposing models that are suitable for distributed algorithms and/or proposing verification techniques that mitigate the state space explosion problem.

In [7], the author has proposed a hybrid approach that combines deductive reasoning and bounded model-checking to formally verify distributed algorithms. At first, the distributed mutual exclusion algorithm of *Ricart and Agrawala* [8] is checked using the calculus of inductive constructions of Coq proof assistant using assertional reasoning. Then, a fault tolerant extension of the same algorithm is implemented using TLA (Temporal Logic of Actions) [9] and checked on a bounded model.

Combining proof and model-checking to validate distributed algorithms is an interesting technique. However, the proposed model deals with a simplified distributed system having few processes with few critical section access requests. Furthermore, the approach still needs expert intervention when proving properties. In addition, the author has not mentioned how to scale up the checking and modelling approaches to deal with complex distributed systems.

In [10], the author proposed the use of a refinement-based construction of distributed algorithms. At first, a starting abstract formal model of a distributed algorithm is built. Then, the model is refined iteratively by adding required behaviours. A verification by construction approach is followed to ensure correctness of distributed algorithms. The work uses Event-B modelling language [11] that allows the gradual development of systems through refinement.

In [12], authors have tackled the challenge of checking correctness of distributed algorithms running on dynamic environments. Their approach proposes a set of reusable patterns that facilitate the specification and verification efforts of distributed algorithms. More precisely, a basic pattern that deals with topological events is first defined. Then, the pattern is extended using the refinement technique to specify particular distributed systems topologies. A leader election distributed algorithm is used to show the efficiency of the approach and a set of proofs associated with the development of the algorithm are presented. The pattern-based approach is implemented using Event-B and the correctness of resulting specification is ensured by a set of automatic and interactive proof obligations (POs).

Despite the importance of gradual verification of distributed algorithms, both [10, 12] works need expert skills in discharging interactive POs. In addition, the authors have not shown how to generalize their approaches to be applied to other distributed algorithms.

In [3], authors have proposed a reusable formal framework for model checking round-based distributed algorithms. To mitigate the space-explosion problem, two reduction techniques were proposed: *partition symerty reduction* and *message order reduction. The partition symmetry technique* exploits the symmetry that arises when processes choose non-deterministic initial value, that is, rather than exploring all possible combinations of processes' initial states, only a subset of initial states are visited. *The message order reduction technique* is applied when the order in which messages are sent or received is irrelevant for the final result of the algorithm. The proposed reduction techniques have been implemented using *Promela* language for the *spin* model checker. Reducing the space state will make it possible to automatically check distributed algorithms. However, the proposed framework suggests reduction techniques that are closely related to only round-based distributed algorithms. Furthermore, to reuse the provided verification template written for the *spin* model checker, users must master *promela* language.

In [13], authors have proposed a framework for programming and checking distributed algorithms. Authors have used *DistAlgo* language for writing distributed algorithms and for expressing safety and liveness properties. The *DistAlgo* language is an object oriented programming language that has a formal semantics and an open source implementation. Authors have proposed the use of *runtime verification* technique for fighting the state space explosion when checking distributed algorithms' properties. An external checker observes messages exchanged between nodes. Then, it verifies safety and liveness properties of the considered algorithms.

The main drawback of the work is its use of a restrictive specification language for expressing checked properties. These properties can only be related to bounded reachability. In addition, *DistAlgo* language has no graphical notations facilitating the reuse of the proposed framework.

In this work, a formal approach for checking distributed algorithms is proposed. In order to tackle the problem of state space explosion, the approach makes use of *runtime verification.* The expected behaviours of the considered distributed algorithms are specified using a high-level Petri net model: Recursive ECATNets. The Verification process consists in examining compliance of extracted execution traces with the recursive ECATNet model of the distributed algorithm. The proposed specification formalism (Recursive ECATNets) is expressive; it allows expressing data flow, control flow as well as advanced synchronisation constraints. Recursive ECATNets are also equipped with graphical notation that facilitates the specification of distributed algorithms.

The recursive ECATNet model and the monitor that checks conformance are implemented using the Maude system. Maude [14] is one of the most efficient implementations of rewriting logic. In fact, we have benefited from the reflective properties of rewriting logic. The Maude built-in descent functions have been used to check whether tackled actions are possible over the recursive ECATNet model.

3. Preliminaries

Recursive ECATNets extend ECATNets with useful modelling features, such as creation of multiple sub-processes and exception handling. Recursive ECATNets can be naturally expressed using the Maude system.

The aim of this section is to present basic concepts on Maude, ECATNets and recursive ECATNets. These concepts will allow a good comprehension of the proposed modelling and checking methods.

3.1. The Maude system

Rewriting logic is a logic of concurrent change that can deal naturally with highly nondeterministic and concurrent computation [15]. Rewriting logic contains an equational subtheory for representing the systems' distributed states. It includes also a rewrite theory for implementing systems' concurrent transitions [16].

Maude is a high-level language for specification and programming in rewriting logic [14]. Maude supports equational programming using its functional modules. Maude's system modules implement concurrent and dynamic systems' evolution. They implement rewriting logic rewrite theory. Maude supports reflection and metaprogramming. This can be achieved using decent functions provided by the module `META-LEVEL`. The Reflection mechanism of Maude provides powerful capabilities. Particularly, it allows to guide and adapt the process of applying rewriting rules at an object level.

ECATNets can be naturally implemented in Maude. In this case, the algebraic equational specification depicts the abstract data type underlining the ECATNet, and the set of rewriting rules implements the dynamic evolution between markings of the ECATNet.

3.2. ECATNets

ECATNets (Extended Concurrent Algebraic Term Nets) are a kind of high-level and algebraic Petri nets, proposed by [17]. ECATNets combine expressiveness and strength of both Petri nets with abstract data types. ECATNets are endowed with a natural implementation in terms of rewriting logic, which has allowed to expand their area of use. ECATNets have been used to model and analyse different kinds of systems. Furthermore, ECATNets allow to describe complex synchronisation constraints and advanced abstract data types [4].

In an ECATNet, places, transitions and arcs are labelled with elements of a multi-set of algebraic terms. Firing rules of ECATNets' transitions establish a clear distinction between enabling marking and consumed tokens. When firing an ECATNet transition, adding and removing tokens take place on a multi-set of ground algebraic terms.

Figure 1 depicts a simple ECATNet having one transition and two places. An ECATNet E is defined by the tuple E = $(Spec, P, T, sort, IC, DT, CT, TC)$, where

- $Spec = (\Sigma, E)$ is the underlining equational algebraic specification of E.
- P is the set of places and T is the set of transitions, with $P \cap T = \emptyset$.
- $sort$ is a function that associates to each place of P a sort. In Figure 1, s and s' are the sorts of places p and p'.
- $IC(p, t)$ is a function that defines a partial condition on input places for firing the transition t.
- $DT(p, t)$ is a function that determines the deleted tokens from input places when firing the transition t.
- $CT(p', t)$ is a function specifying the set of tokens to be created in the output places when firing the transition t.
- $TC(t)$ is a function that expresses further conditions for firing the transition t.

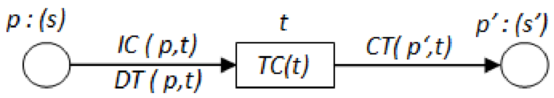


Figure 1: A one transition ECATNet

3.3. Recursive ECATNets

Recursive ECATNets (abbreviated RECATNets) [18] extend ordinary ECATNets with the concept of recursion. RECATNets are well-suited for modelling concurrent systems characterized by dynamic creation and deletion of threads and sub-processes.

In a RECATNet two types of transitions can be distinguished :

- *Elementary transitions* that describe ordinary actions, and allow the termination of running sub-processes. Elementary transitions are graphically depicted by simple rectangles, and
- *Abstract transitions* that are able to dynamically create new child processes that start their execution with a specific initial marking. Abstract transitions are graphically represented by double border rectangles. Their starting marking is mentioned in a rectangle beside the transition.

The space state generated by a RECATNet is a tree of processes (an extended marking). Each sub-process has a set of termination markings that indicate the states in which the sub-process terminates. When a process ends, it terminates all of its child processes and creates in the father process the output tokens of the abstract transition which has created them.

Elementary transitions of RECATNets are fired in the same way as for ECATNets. However, the outgoing arcs of an abstract transition are labelled with an integer index associated with a termination marking. When a sub-process reaches a termination marking, it generates tokens only in the output place linked by an edge having the same index as the index of the termination marking.

The events that can take place in a RECATNet are:

- Firing an abstract transition, and creating at the same time a sub-process,
- Firing an elementary transition that can end a running sub-process (executing a cut step),
- Reaching a termination marking and producing tokens in output places of the creating abstract transition.

A RECATNet is defined by the tuple $Rec = (Ect, I, \Upsilon, ICT)$, where

- $Ect = (Spec, P, T, sort, IC, DT, CT, TC)$, is the underlying ECATNet, with $T = T_{el} \cup T_{abs}$ is the set of abstract and elementary transitions,
- I is a finite set of termination indexes,
- Υ is a family indexed by I of final markings,
- $ICT : P \times T_{abs} \times I \rightarrow T_{\Sigma}(X)$ is a function specifying the tokens created in output places of abstract transitions.

4. Runtime verification of distributed algorithms using recursive ECATNets

Unlike centralized systems, distributed systems have no common physical clock and memory. This will complicate the processes synchronisation task. Furthermore, partial failure, unpredictable communication delays and difference in node processing speed make distributed algorithms managing and control more difficult and challenging.

The aim of this section is to detail the proposed runtime verification of distributed algorithms. The process is organized in two steps: specifying the desired behaviour of the distributed algorithm as a RECATNet model, and then generating the monitor that extracts the necessary information for monitoring and checks whether the the runtime behaviour of the distributed algorithm conforms to the RECANet model.

This section starts by describing the adopted system model and the chosen case study. Then, the proposed RECATNet model is depicted. Lastly, we show how to check at runtime the distributed algorithm regarding the RECATNet model.

4.1. The system model and case study

The addressed problem in this work is a synchronization problem: the mutual exclusion. The issue is to ensure that, at any moment, at most one process is authorized to use a resource.

Two kinds of distributed algorithms were proposed to ensure the exclusive access to a shared resource: token-based algorithms and non-token-based (permission-based) algorithms [19]. In permission-based algorithms, a node that needs to use the critical resource must at first receive permission from other nodes. Contrariwise, when using token-based algorithms, the right to access a critical resource is conditioned by obtaining the token.

The case study algorithm of this work is the well-known token-based mutual exclusion algorithm of *Suzuki and Kasami* [20]. In this algorithm, a unique token (a special message) is shared among all the nodes of the distributed system. A node is allowed to access the critical section only if it possesses the token. When a site wants to use the critical resource and it has not the token, it broadcasts request messages to all other sites. A site releases the token only if it has completed its use of the critical resource. When a site terminates its execution of the critical section, it sends the token to another requesting site.

The algorithm of *Suzuki and kasami* provides several advantages: it has a low message overhead, it offers a clear and intuitive approach for achieving mutual exclusion, and it demonstrates robustness against failure events within a distributed system.

Through this paper, the following conventions are adopted:

- The system is composed of n nodes (sites).
- It is assumed that only one process runs on each site. Thus, the terms site, node and process are used interchangeably.
- The nodes are linked by reliable communication channels.
- The processes do not share the memory or the physical clock. So, they only communicate by exchanging messages.
- The communication delays are finite.
- The distributed system is conceptually fully connected; every site in the system can send messages to any other site of the system.
- Nodes use the critical resource for fixed durations.

4.2. RECATNet model of *Suzuki and kasami* algorithm

The expected behaviour of the distributed algorithm is expressed using a RECATNet. The adopted modelling approach makes use of several threads. In this case, a thread may send requests to access the critical section, while another thread responds to messages received from other sites. The RECATNet model associated to a distributed system executing *Suzuki and Kasami* distributed algorithm is shown in Figure 2.

Let us first mention that initially, the place `p1` contains `n` tokens, each token is associated to a site. Tokens of the place `p1` have a well defined structure: any token has a component of the sort natural numbers (`Nat`) that represents the site identifier, the second and the third components mention the state of the site and whether the site has the right to use the critical resource.

Informally the model of Figure 2 can be explained as follows. At first, the transition `initialize` is fired, it consumes all the tokens from the place `p1` and produces copies of these tokens in the output places `p2` and `p3`. This will allow for each site to respond to messages received from other sites, and to send requests for acquiring the critical resource.

When the place `p2` becomes marked, a site can receive requests from other sites by executing the transition `t4`. Then, the site can either emit the token (by firing the transition `t5`) or going back (executing the transition `t3`) if it has not the token or if it is needing the resource. In addition, when the place `p2` is marked, a site can receive the token from other sites by firing the transition `t2`.

Moreover, the place `p3` allows nodes to become requesters. If the site is already requester and has the token, then it directly uses the resource (it fires the transition `t6`). Otherwise, the site executes a call to acquire the resource through launching the abstract transition `Request-CS`.

The execution of the abstract transition `Rquest-CS` dynamically creates a new sub-process that starts its execution with `n-1` tokens in place `p6`. This process is responsible for sending requests to the remaining sites to obtain the token.

The execution of the sub-process launched by the transition `Rquest-CS` is interrupted automatically when the elementary transition `t2` (receipt of the token) is fired with the index <0>. The execution of this cut step also produces a token in place `p3`. This will subsequently allow the access to the critical resource.

The place `p5` contains the token associated to the site using the critical resource. When the abstract transition `Realeas-CS` is fired, a sub-process for releasing the resource is launched. This process starts with a token at the place `p8`. At first, the process leaving the critical section goes to the outside state (firing of the transition `t9`). Then, if there is a pending request from another site, the token will be passed to it (firing the transition `t11`). Otherwise, the token will be maintained (execution of the transition `t10`).

The end of a call to release the resource is mentioned by the presence of a token in place `p10` (see termination state Υ_1). During this termination, a token is produced in place `p3`. This allows a site to become requester of the critical resource.

The RECATNet model of the Figure 2 is implemented using Maude. Places identifiers are specified by algebraic ground terms of the sort `PlaceId`.

```
ops localVar p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 : -> PlaceId .
```

Additionally, the states through which sites pass, as well as the number of sites are specified by algebraic constructor terms:

```
ops InCs Reqstr outside : -> State [ctor] . op nbS : -> Nat [ctor] .
```

Then, one equation is needed to define overall Maude RECATNet model. In fact, the equation defines the marking of the RECATNet, it associates each place identifier with a set of tokens.

```
eq inits = (p1,'tk 0 'tk 1 'tk 2...) (p2,noneTk) (p3,noneTk) (p4,
  noneTk) (p5, noneTk) (p6, noneTk) (localVar, outside TkAv 0 outside
  tkNav 1 outside tkNav 2 ... ) < "" > .
```

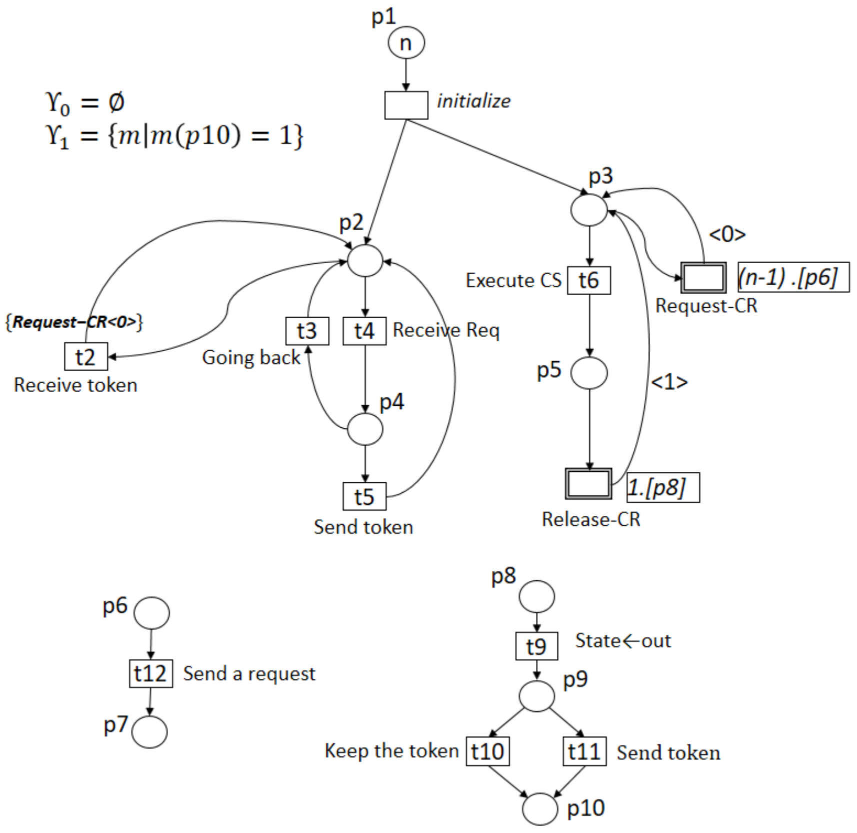


Figure 2: The RECATNet model specifying the expected behaviour of *Suzuki and Kasami* distributed algorithm

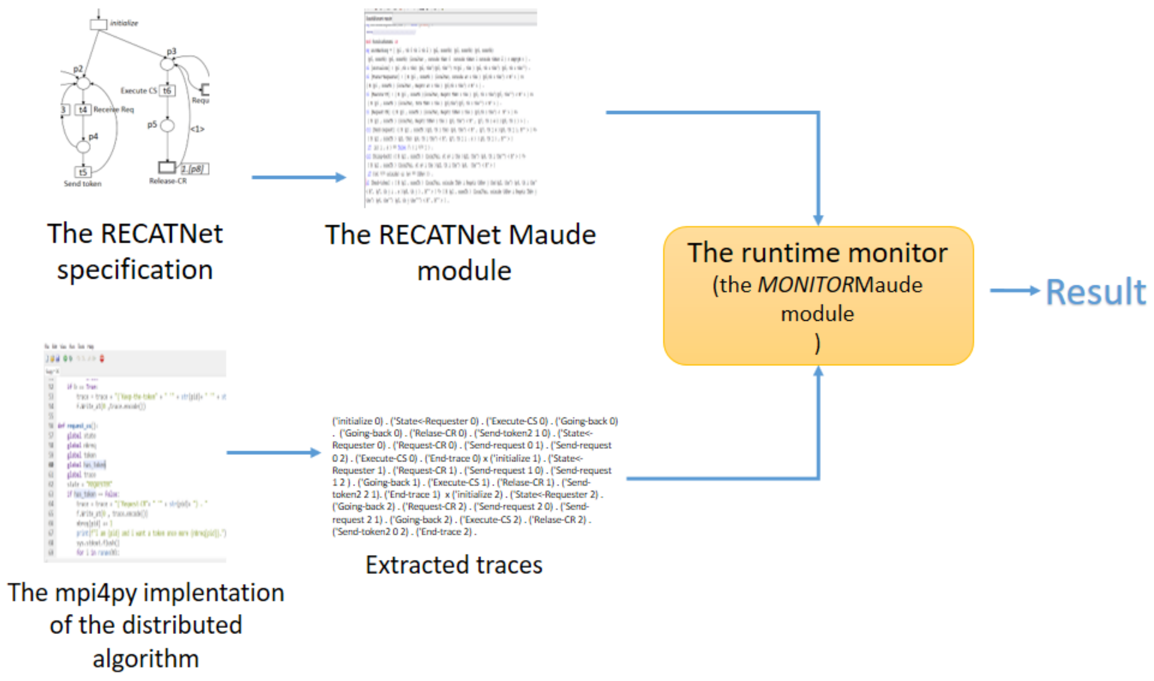
The last implementation step of *Zusuki and Kasami* algorithm RECATNet is to associate to each transition a Maude rewriting rule. The left-hand side of each rule is a term that represents conditions on input places to fire the transition. If conditions of the left-hand side are matched by the RECATNet marking, then the marking will be updated according the right-hand side term of the rule. The condition part of each rewrite rule implements further constraints on input marking to fire the RECATNet transitions.

The following code shows the rewriting rules associated with the transition that receives a request and the transition for requesting the critical resource.

```
crl [Receive-request]:{M(p2,'tk i tks)(p4,tks')<M',(p7,'tk j s)(p8,'tk
   j),M''>} => {M(p2, tks)(p4,'tk i tks')<M',(p7,'tk j i.s)(p8, 'tk j),
  M''>} if in(i,s)==false/\(i =/= j).
rl [Reques-Cs] :{ M (localVar, Reqstr tkNav i tks)(p3,'tk itks')<M'>}=>
   {M (localVar,Reqstr tkNav itks)(p3,tks')<M',(p7,'tk i e-1)(p8,'tk i
  (INF))>} .
```

4.3. Runtime monitoring of *Zusuki and Kasami* algorithm against the RECATNet model

Figure 3 shows all the elements involved in the monitoring task. The monitor is a Maude module that accepts as input traces of events extracted from the considered distributed algorithm execution. As a second input, the monitor accepts the Maude implementation of the RECATNet model.



```
('initialize 0) . ('State<-Requester 0) . ('Execute-CS 0) . ('Going-back 0)
. ('Going-back 0) . ('Relase-CR 0) . ('Send-token2 1 0) . ('State<-
Requester 0) . ('Request-CR 0) . ('Send-request 0 1) . ('Send-request
0 2) . ('Execute-CS 0) . ('End-trace 0) x ('initialize 1) . ('State<-
Requester 1) . ('Request-CR 1) . ('Send-request 1 0) . ('Send-request
1 2 ) . ('Going-back 1) . ('Execute-CS 1) . ('Relase-CR 1) . ('Send-
token2 2 1). ('End-trace 1)  x ('initialize 2) . ('State<-Requester 2) .
('Going-back 2) . ('Request-CR 2) . ('Send-request 2 0) . ('Send-
request 2 1) . ('Going-back 2) . ('Execute-CS 2) . ('Relase-CR 2) .
('Send-token2 0 2) . ('End-trace 2) .
```

initialize
p2
p3
Execute CS
t6
t4
Receive Req
Requ
p5
p4
<1>
t5
Send token
Release-CR
The RECATNet specification
The RECATNet Maude module
The runtime monitor
(the *MONITOR*Maude module
)
Result
The mpi4py implentation of the distributed algorithm
Extracted traces

Figure 3: The RECATNets model specifying the expected behaviour of *Suzuki and Kasami* algorithm

The algorithm under scrutiny is implemented using MPI (Message Passing Interface) for Python which is a standardized communication protocol widely used for implementing distributed applications. Each node of the distributed system encapsulates an events catching mechanism that records important events as an execution trace. In fact, we have adopted an asynchronous online monitoring. That is, while the distributed algorithm is checked during its execution, the monitor receives the system's trace of events asynchronously without suspending any node of the system. Then, the monitor analyses traces at its own pace. This checking strategy allows to significantly reduce overhead.

The Maude monitor module benefits from the metaprogramming capabilities of Maude. The monitor module uses the decent function `metaXapply` to check whether the transitions of the RECATNet model (implemented as rewriting rule) corresponding to the traces' events are fireable. In this case, the monitor passes to the following events and update the RECATNet model according to the fired transition. If no event is possible in the RECATNet model, then the monitor concludes that the events traces violate the correctness of the analysed distributed algorithm.

The following listing includes a slightly modified version of the MONITOR Maude module. The module declares the operation Verify that accepts as a first parameter the meta representation of a term denoting the marking of the *Suzuki and kasami* algorithm's RECATNet model. The second parameter of the operation Verify is a term that gathers all traces. The module MONITOR uses two rewriting rules to check the conformance of the finite execution traces against the RECATNet model. The first rule checks execution of events having one parameter, whereas the second checks execution of events having two parameters.

```
mod MONITOR is
including TRACES .
protecting META-LEVEL .
*** declaring used variables
var N : Nat .  vars n n' : Nat .          var T : Term .
var qd : Qid . vars Tr Tr' Tr'' : Traces . var Result? : [Result4Tuple].
*** declaring paramaters of the operation that check execution of traces'
  events
op Verify : Term Traces -> Term .
*** implmenting the operation Verify using two rewriting rules
crl Verify(T , (qd n) . Tr x Tr'x Tr'')
=> if Result?::Result4Tuple then Verify(getTerm(Result?), TrxTr'xTr'')
else Verify(T , Tr'xTr''x(qd n) . Tr) fi
if Result? := metaXapply(upModule('Suzuki, false), T, qd, 'i:Nat <- upTerm
  (n), 0, unbounded, 0) .
crl Verify( T, ( qd n n' ) . Tr x Tr' x Tr'')
=> if Result?::Result4Tuple then Verify(getTerm(Result?), Trx Tr'xTr'')
else Verify(T , Tr' x Tr'' x ( qd n n' ) . Tr) fi
if Result? := metaXapply(upModule('Suzuki, false), T, qd, 'i:Nat <- upTerm
  (n'); 'j:Nat <- upTerm(n), 0, unbounded, 0) .
      endm
```

The following Maude code shows how to use the operation Verify to check at runtime the correctness of *Suzuki and kasami* algorithm. In fact, we have checked an execution of the algorithm that involves three nodes having three traces. In order to check conformance, users need only to execute the Maude search command that checks whether it is possible to reach a state in which all traces events where applied on the RECATNet model, i.e. reaching the term : ('End-trace 0) x ('End-trace 1) x ('End-trace 2) as traces, which is the case for the following code.

```
=================
Maude> search in MONITOR : Verify(upTerm(inits),Traces) =>* Verify(T,Tr)
  such that Tr == ('End-trace 0) x ('End-trace 1) x ('End-trace 2) .
=================
Solution 1 (state 46)
states: 47 rewrites: 515 in 5811094365ms cpu (1194ms real) (0 rewrites/
  second)
T --> ''{_'}['_<_>['__[''(_',_')['localVar.PlaceId,'__['___['InCs.State, '
  TkAv.Availabilaty,'0.Zero],'___['outside.State,'tkNav.Availabilaty ... Tr
  --> ('End-trace 0) x ('End-trace 1) x ('End-trace 2)
```

```
No more solutions.
=================
```

5. Conclusion

The motivation of the proposed specification and modelling approach presented in this paper comes from the need to cope with the state space explosion problem when formally checking distributed algorithms. Hence a runtime verification approach is used in this work to verify the distributed algorithms correctness.

The desired behaviour of a distributed algorithm is expressed using RECATNets. This has enabled a natural modelling of multithreaded and interruptible computation. The proposed modelling and checking approach was implemented using Maude language, especially by leveraging its metalevel capabilities. The modelling and checking approach presented in this section can be easily reused to check the correctness of other distributed algorithms.

As a future extension of the work, we plan to extend the proposed model to take in consideration timed and fault-tolerant behaviour of distributed algorithms. We intend also to explore how to achieve a choreographed runtime checking where monitors are deployed across more than one node. This is more in harmony with distributed algorithms principles. Besides, implementing a graphical editor for creating distributed algorithms RECATNets models and then translating them into Maude programs will be very useful.

Acknowledgment

This work was partially supported by the LABEX-TA project MeFoGL: "Méthode Formelles pour le Génie Logiciel"

References

[1] M. Van Steen, A. S. Tanenbaum, A brief introduction to distributed systems, Computing 98 (2016) 967–1009.

[2] L. Lamport, A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." á cacm, 1992.

[3] R. Barbosa, A. Fonseca, F. Araujo, Reductions and abstractions for formal verification of distributed round-based algorithms, Software Quality Journal 29 (2021) 705–731.

[4] A. Hicheur, K. Barkaoui, N. Boudiaf, Modeling workflows with recursive ecatnets, in: 2006 Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2006, pp. 389–398. doi:10.1109/SYNASC.2006.52.

[5] A. Francalanza, J. A. Pérez, C. Sánchez, Runtime verification for decentralised and distributed systems, Lectures on Runtime Verification: Introductory and Advanced Topics (2018) 176–210.

[6] F. Fakhfakh, M. Tounsi, M. Mosbah, A. H. Kacem, Formal verification approaches for distributed algorithms: A systematic literature review, in: R. J. Howlett, L. C. Jain, Z. Popovic,

D. B. Popovic, S. N. Vukosavic, C. Toro, Y. Hicks (Eds.), Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 22nd International Conference KES-2018, Belgrade, Serbia, 3-5 September 2018, volume 126 of *Procedia Computer Science*, Elsevier, 2018, pp. 1551–1560. doi:10.1016/j.procs.2018.08.128.

[7] E. Shishkin, Construction and formal verification of a fault-tolerant distributed mutual exclusion algorithm, in: N. Chechina, S. L. Fritchie (Eds.), Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang, Oxford, United Kingdom, September 3-9, 2017, ACM, 2017, pp. 1–12. doi:10.1145/3123569.3123571.

[8] G. Ricart, A. K. Agrawala, An optimal algorithm for mutual exclusion in computer networks, Communications of the ACM 24 (1981) 9–17.

[9] L. Lamport, Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers, Addison-Wesley, 2002.

[10] D. Méry, Refinement-based construction of correct distributed algorithms, in: 2021 Second International Conference on Information Systems and Software Technologies (ICI2ST), IEEE, 2021, pp. 46–53.

[11] J.-R. Abrial, Modeling in Event-B: system and software engineering, Cambridge University Press, 2010.

[12] F. Fakhfakh, M. Tounsi, M. Mosbah, Modeling and proving distributed algorithms for dynamic graphs, Future Gener. Comput. Syst. 108 (2020) 751–761. doi:10.1016/j.future.2020.03.003.

[13] Y. A. Liu, S. D. Stoller, Assurance of distributed algorithms and systems: Runtime checking of safety and liveness, in: J. Deshmukh, D. Nickovic (Eds.), Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings, volume 12399 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 47–66. doi:10.1007/978-3-030-60508-7_3.

[14] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martı-Oliet, J. Meseguer, R. Rubio, C. Talcott, Maude manual (version 3.1), SRI International University of Illinois at Urbana-Champaign http://maude. lcc. uma. es/maude31-manual-html/maude-manual. html (2020).

[15] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, J. F. Quesada, Maude: Specification and programming in rewriting logic, Theoretical Computer Science 285 (2002) 187–243.

[16] J. Meseguer, Twenty years of rewriting logic, The Journal of Logic and Algebraic Programming 81 (2012) 721–781.

[17] M. Bettaz, M. Maouche, How to specify non determinism and true concurrency with algebraic term nets, in: Recent Trends in Data Type Specification, Springer, 1991, pp. 164–180.

[18] K. Barkaoui, A. Hicheur, Towards analysis of flexible and collaborative workflow using recursive ecatnets, in: Business Process Management Workshops: BPM 2007 International Workshops, BPI, BPD, CBP, ProHealth, RefMod, semantics4ws, Brisbane, Australia, September 24, 2007, Revised Selected Papers 5, Springer, 2008, pp. 232–244.

[19] M. Raynal, Distributed Algorithms for Message-Passing Systems, Springer, 2013. doi:10.1007/978-3-642-38123-2.

[20] I. Suzuki, T. Kasami, A distributed mutual exclusion algorithm, ACM Trans. Comput. Syst. 3 (1985) 344–349. doi:10.1145/6110.214406.