Timed-Rebeca Schedulability and Deadlock-Freedom Analysis Using Floating-Time Transition System

Ehsan Khamespanah[1]
Zeynab Sabahi[1]
Ramtin Khosravi[1]
Marjan Sirjani [2,1]
Mohammad-Javad Izadi[1]

[1] **University of Tehran, Iran**
[2] **Reykjavik University, Iceland**

AGERE!@SPLASH 2012
21-22 October 2012 - Tucson, Arizona, USA

Outline

- Motivation (and where do we stand)
- Timed Rebeca – the language
- Semantics and Floating Time Transition System
- Schedulability and Deadlock Freedom
- Conclusion and Future Work

Systems now-a-days

- Concurrent
- Event-based
- Based on message passing (distributed or not)

- Timing is becoming more and more important

Questions to answer

- Are our systems correct?
- Are they performing well enough?

Applications: correctness, performance, and more ...

- A correct distributed ticket service
- A correct distributed battleship game
- A reliable control program for Toyota brake system
- A sound program for the traffic lights on a crossing
- A deadlock-free routing algorithm in a network

Engineers need Models

A good model has to be:

- Analyzable:
  - have a solid basis
- Usable:
  - capture all we need,
  - understandable for the developer

Tradeoff!

Different models

- Timed Automata
- Timed Petri nets
- Real-time Maude
- Ptolemy
- Timed Rebeca

Rebeca: The Modeling Language

- **Re**active o**bjec**t l**a**nguage

  (Sirjani-Movaghar, Sharif U. of Technology, 2001)

- Imperative Actor-based language
  - Concurrent reactive objects (OO)
  - Java like syntax
  - Simple core

  - Tools do not support dynamic creation and topology

Rebeca models

- Communication:
  - Asynchronous message passing: non-blocking send
  - Unbounded message queue for each rebec
  - No explicit receive
- Computation:
  - Take a message from top of the queue and execute it atomically (macro-step)
  - Event-driven

More on Rebeca …

- Model checking support
- Compositional verification
- Symmetry and partial order
- Slicing

- Used to model check SystemC
- Now working on MPI (ongoing)
- Extended for self-adaptive systems
- Product line software

References

- M. Sirjani, A. Movaghar, and M.R. Mousavi, **Compositional Verification of an Actor-Based Model for Reactive Systems**, in Proceedings of Workshop on Automated Verification of Critical Systems (AVoCS'01), Oxford University, April 2001.
- M. Sirjani, M. M. Jaghoori, **Ten Years of Analyzing Actors: Rebeca Experience**, LNCS 7000, pp. 20-56, 2011.
- M. Sirjani, A. Movaghar, A. Shali, F.S. de Boer, **Modeling and Verification of Reactive Systems using Rebeca**, Fundamenta Informaticae, Volume 63, Number 4, ISSN 0169-2968, pp. 385-410, 2004.
- M . M. Jaghoori, M. Sirjani, M. R. Mousavi, E. Khamespanah, A. Movaghar, **Symmetry and Partial Order Reduction Techniques in Model Checking Rebeca**, Acta Informatica, Volume 47, Issue 1, pp. 33-66, 2009.
- .....

Rebeca Formal Modeling Language

Rebeca

View Edit History Print

Rebeca (Reactive Objects Language) is an actor-based language with a formal foundation, designed in an effort to bridge the gap between formal verification approaches and real applications. It can be considered as a reference model for concurrent computation, based on an operational interpretation of the actor model. It is also a platform for developing object-based concurrent systems in practice.

Besides having an appropriate and efficient way for modeling concurrent and distributed systems, one needs a formal verification approach to ensure their correctness. Rebeca is supported by Rebeca Verifier tool, as a front-end, to translate the codes into existing model-checker languages and thus, be able to verify their properties. Modular verification and abstraction techniques are used to reduce the state space and make it possible to verify complicated reactive systems.

Rebeca is an actor-based language for modeling and verification of reactive systems. Modeling a system in Rebeca requires one to specify reactive-object templates and a finite set of object instances that run in parallel. Properties can be specified in temporal logic. Different approaches are proposed for verifying correctness of these properties.

The key features of Rebeca are:

- using actor-based concepts for the specification of reactive systems and their communications;
- introducing components as an additional structure for verification purposes;
- providing a formal semantics for the model and components, comprising their states, communications, state transitions, and the knowledge of accessible interfaces;
- using different abstraction techniques which preserve a set of behavioral specification in temporal logic, and reduce the state space of a model, making it more suitable for model checking techniques;
- establishing the soundness of these abstraction techniques by proving a weak simulation relation between the constructs;
- applying a compositional verification approach, using the specified abstraction techniques;
- translating Rebeca models into target languages of existing model checkers, enabling model checking of open, distributed systems.
- direct model checking using RMC.

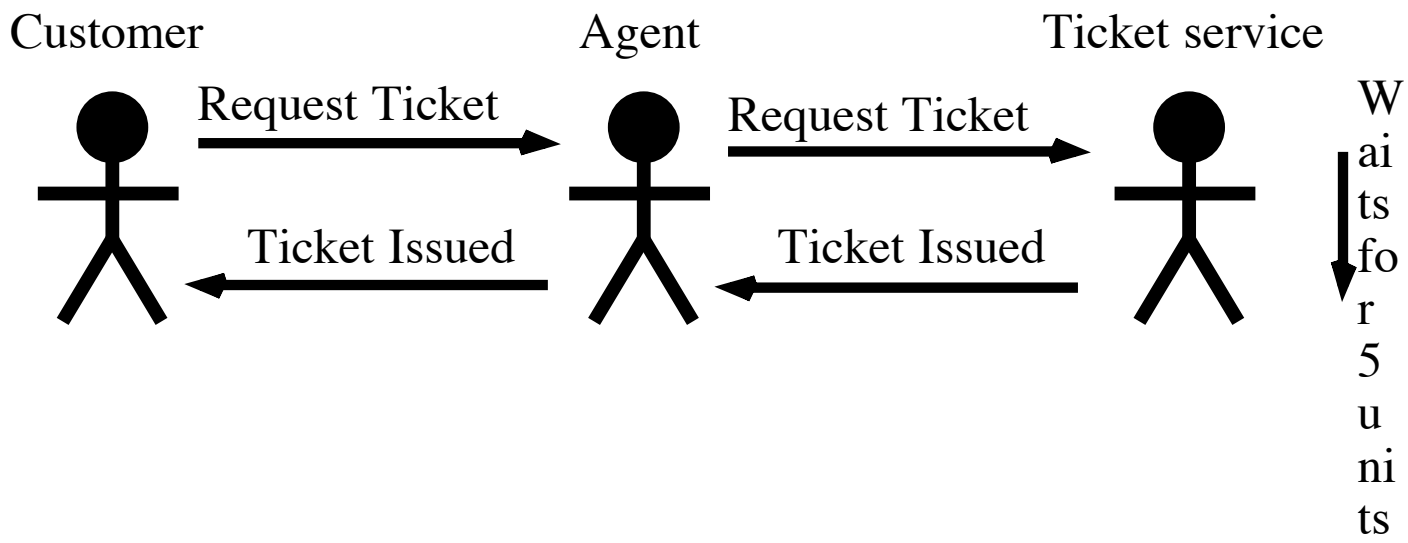
Rebeca, is inspired by the actors paradigm, but goes beyond it by adding the concept of components and the ability to analyze a group of reactive objects as a component. Also, we have classes that reactive objects are instantiated from. Classes serve as templates for state, behavior, and the interface access; adding reusability in both modeling and verification process.

Timed-Rebeca

- An extension of Rebeca for real time systems modeling
  - Computation time (delay)
  - Message delivery time (after)
  - Message expiration (deadline)
  - Periods of occurence of events (after)

Timed-Rebeca Example

- Example of ticket service system
- A Customer wants to buy a ticket
  - There is time constraint for issuing ticket



Ticket Service model

```
reactiveclass TicketService {
  knownrebecs {
    Agent a;
  }
  statevars {
    int issueDelay;
  }
  msgsrv initial(int myDelay) {
    issueDelay = myDelay;
  }
  msgsrv requestTicket() {
    delay(issueDelay);
    a.ticketIssued(1);
  }

ass Agent {
ebecs {
tService ts;
mer c;
  }
  msgsrv requestTicket() {
    ts.requestTicket() deadlin
  }
```

```
  msgsrv ticketIssued(byte id) {
    c.ticketIssued(id);
  }
}
reactiveclass Customer {
  knownrebecs {
    Agent a;
  }
  msgsrv initial() {
    self try();
  }
ry() {
estTicket();
  }
ticketIssued(byte id) {
ry() after(30);
  }
}
in {
gent a(ts, c):();
icketService ts(a):(3);
ustomer c(a):();
}
```

Act
its

Communication delay or periodic tasks

Asynchronous message sending

Time progress because of computation delay

Deadline for the message release

Instances of three different actors

Timed-Rebeca Semantics

- Formal semantics given as SOS rules
- The main rule is the schedular rule:

$$\frac{(\sigma_{r_i}(m), \sigma_{r_i}[rtime = max(TT, \sigma_{r_i}(now)), [\overline{arg} = \overline{v}], sender = r_j], Env, B) \xrightarrow{\tau} (\sigma'_{r_i}, Env', B')}{(\{\sigma_{r_i}\} \cup Env, \{(r_i, m(\overline{v}), r_j, TT, DL)\} \cup B) \rightarrow (\{\sigma'_{r_i}\} \cup Env', B')} C$$

The schedular and progress of time

- The scheduler picks up messages from the bag and execute the corresponding methods.
- ***delay*** statements change the value of the current local time, ***now***, for the considered rebec.
- The time tag for the message is the current local time (***now***), plus value of the ***after***
- The scheduler picks the message with the

Floating-Time Transition System: FTTS

- A state contains
  - Rebecs' state variables valuation
  - Rebecs' message bags
  - Local time of rebecs
- Example of initial state
  - its rebecs have initial message in their message bags
  - Local times are 0

S_0		
a	State vars:	
	Messasge Bag:	$[(initial, 0, \infty)]$
	Now:	0
ts	State vars:	issueDelay=?
	Messasge Bag:	$[(initial, 0, \infty)]$
	Now:	0
c	State vars:	
	Messasge Bag:	$[(initial, 0, \infty)]$
	Now:	0

Floating-Time Transition System: rebecs with different local times

- Called floating-time because of different local times of rebecs
  - There is no global time in each state
- Example of a state in which the rebecs are in different local times

S_{15}		
a	State vars:	
	Messasge Bag:	[ ]
	Now:	6
ts	State vars:	issueDelay=3
	Messasge Bag:	[ ]
	Now:	6
c	State vars:	
	Messasge Bag:	$[(ts.ticketissued, 6, \infty)]$
	Now:	0

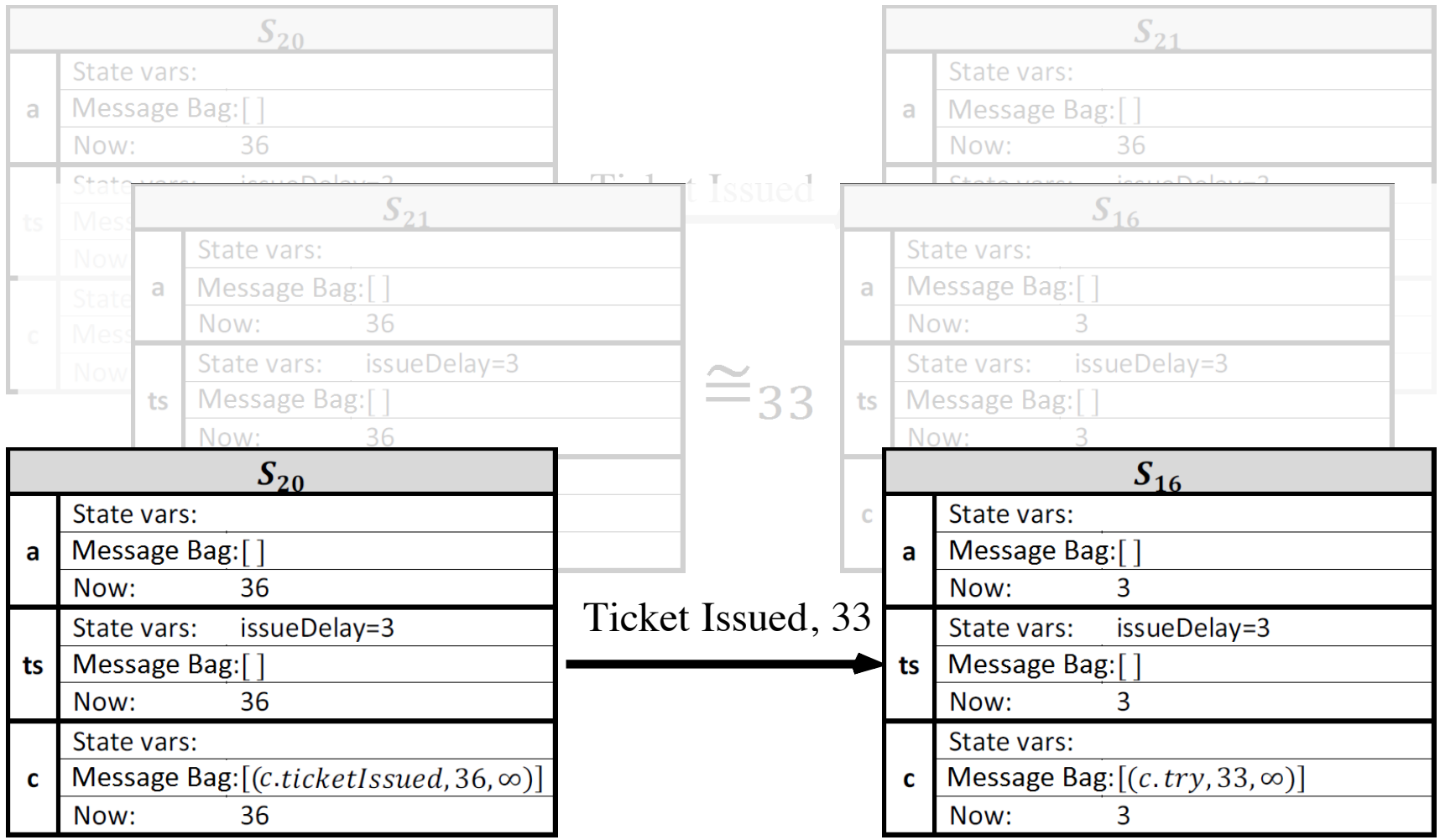
Transition in FTTS

- Releasing and executing a messages
  - Assign new values to state variables
  - Sending some new messages
  - Changing the local time of the rebec because of the delay statement

Bounded Floating-Time Transition System

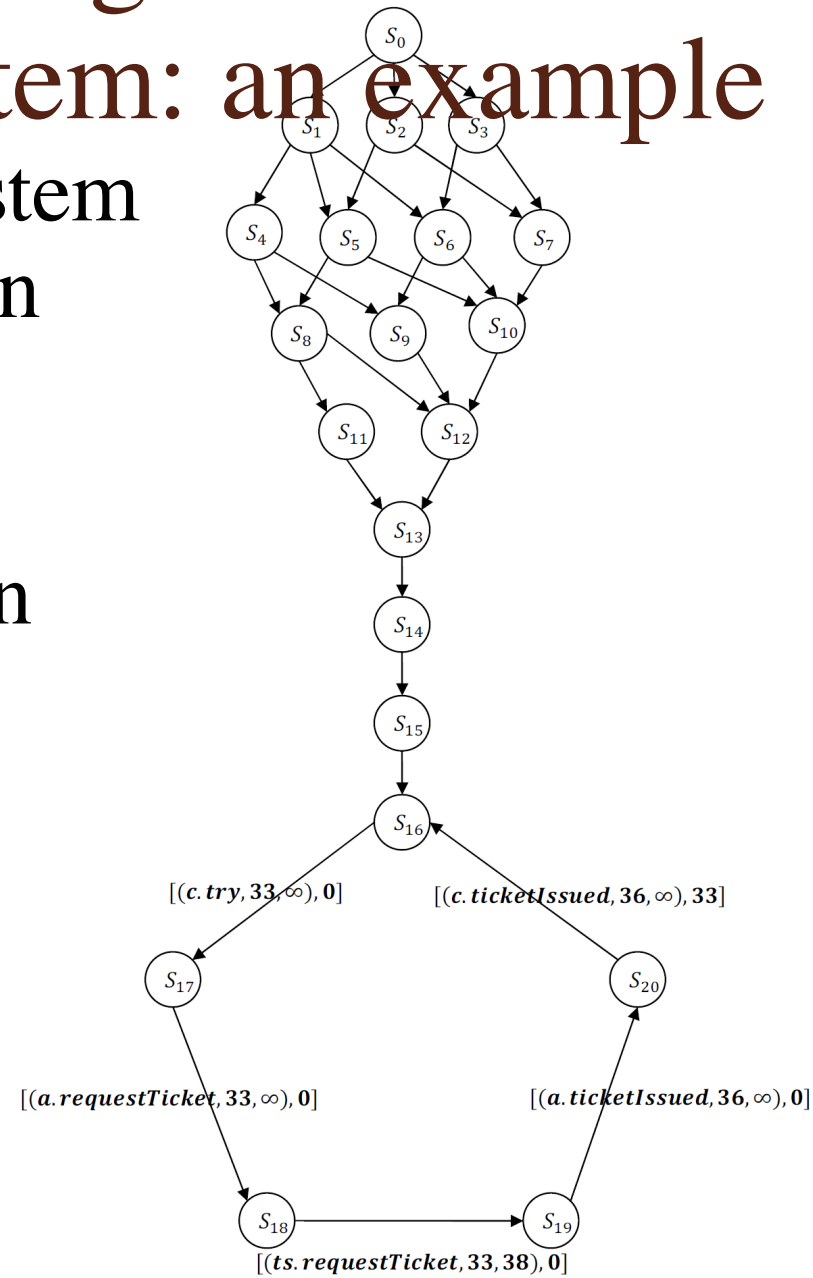
- A new notion of state equivalence by shifting the local times of rebecs
- Time in Timed-Rebeca models is relative
  - Uniform shift of time to past or future has no effect on the execution of statements

Bounding the Floating-Time Transition System



Bounded Floating-Time Transition System: an example

- Ticket service system complete transition system
- A shift-time transition, between states 16 and 20



Bounded FTTS and FTTS

- Bounded floating-time transition system and floating-time transition system are bisimilar – (if we define a correct equivalency relation for the *timing values)*

- Two following conditions should be satisfied for two bisimilar states s and s’

1. If s has a successor state q then s’ has a successor state q’ which q and q’ are bisimilar and vice versa
2. Labels of s and s’ are the same

- Condition 1 holds because
  - The (result of) execution of a message is the same in FTTS and Bounded FTTS
  - The message bag contents are the same – with a shift in timing values

Deadlock and Deadline-missed

- Deadlock happens when the message bag is empty: same for both
- Deadline is missed when *now* is greater than *deadline:* in Bounded FTTS we shift both *now* and *deadline* consistently.

Deadlock and schedulability check

- We keep the relative distance between values of all the timing values of each state (relative timing distances are preserved)
- Deadlines are set relatively so time shift has no effect on deadline-miss
- For checking “deadline missed” and “deadlock” relative time is enough

State space reduction: a simple Timed-Rebeca Model

```
reactiveclass RC1 (3) {
  knownrebecs {
    RC2 r2;
  }
  {
    m1();
  }
  msgsrv
    delay
    r2.m2
    delay
    r2.m
    self.m
  }
}
```

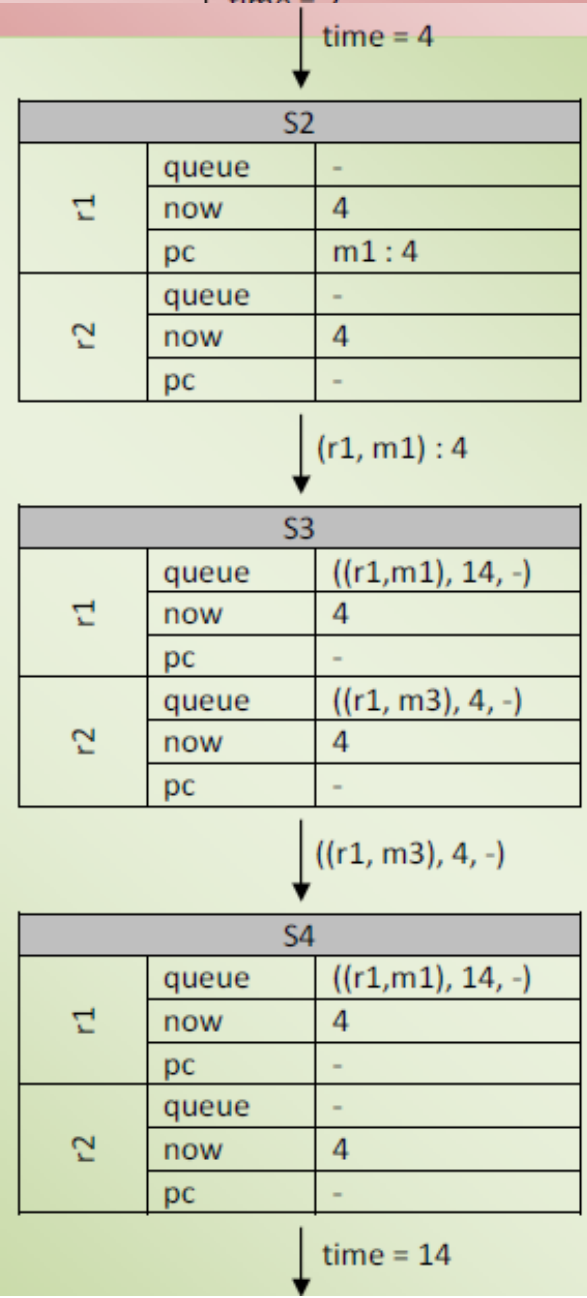
```
reactiveclass RC2 (4) {
  knownrebecs {
    RC1 r1;
  }
  RC2() { }
  2() { }
  3() { }
  RC1 r1(r2):();
```

Line number as program counter

```
msgsrv m1() {
1       delay(2);
2       r2.m2();
3       delay(2);
4        r2.m3();
5       self.m1() after (10);
  }
```

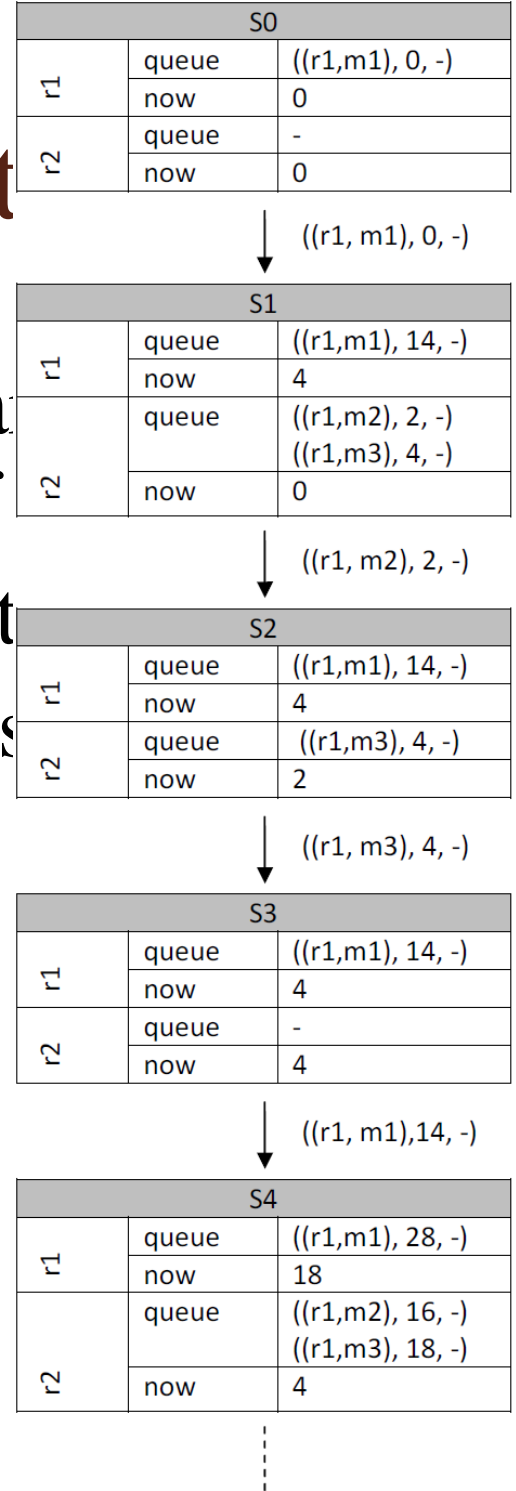
of The

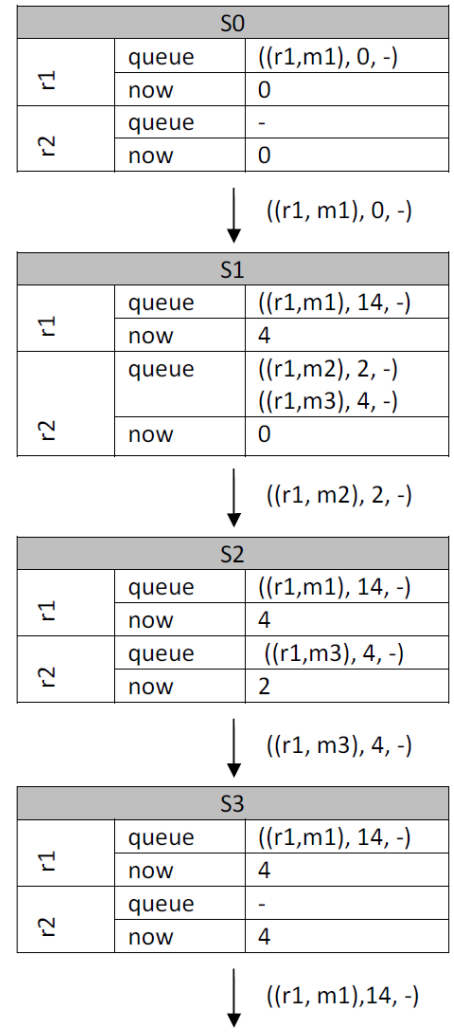
time = 4


time = 4
S2
r1
queue -
now 4
pc m1 : 4
r2
queue -
now 4
pc -
(r1, m1) : 4
S3
r1
queue ((r1,m1), 14, -)
now 4
pc -
r2
queue ((r1, m3), 4, -)
now 4
pc -
((r1, m3), 4, -)
S4
r1
queue ((r1,m1), 14, -)
now 4
pc -
r2
queue -
now 4
pc -
time = 14

FTTS of t… model

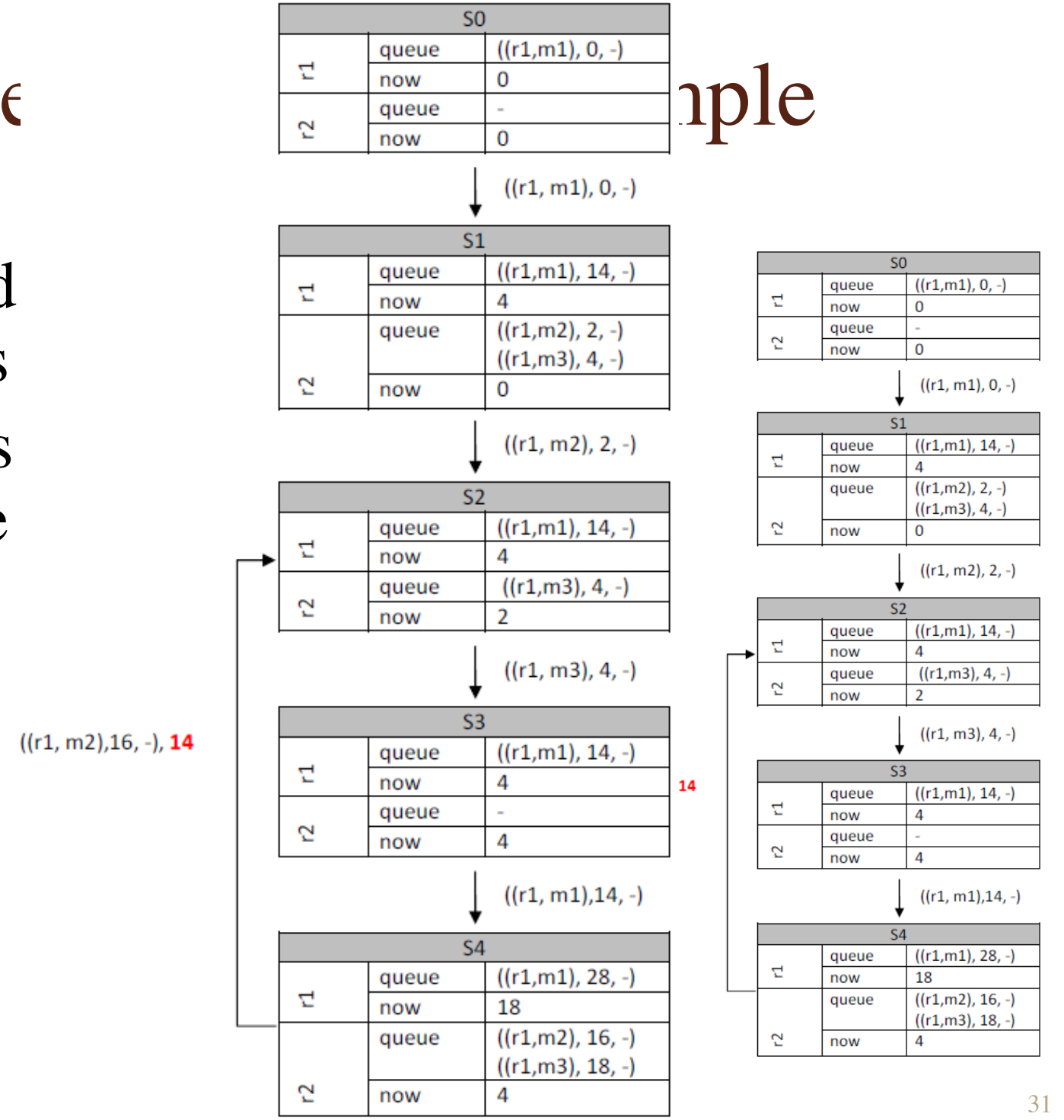
- Four states a… or one round of …
- PCs are omit…
- Unbounded s… generated

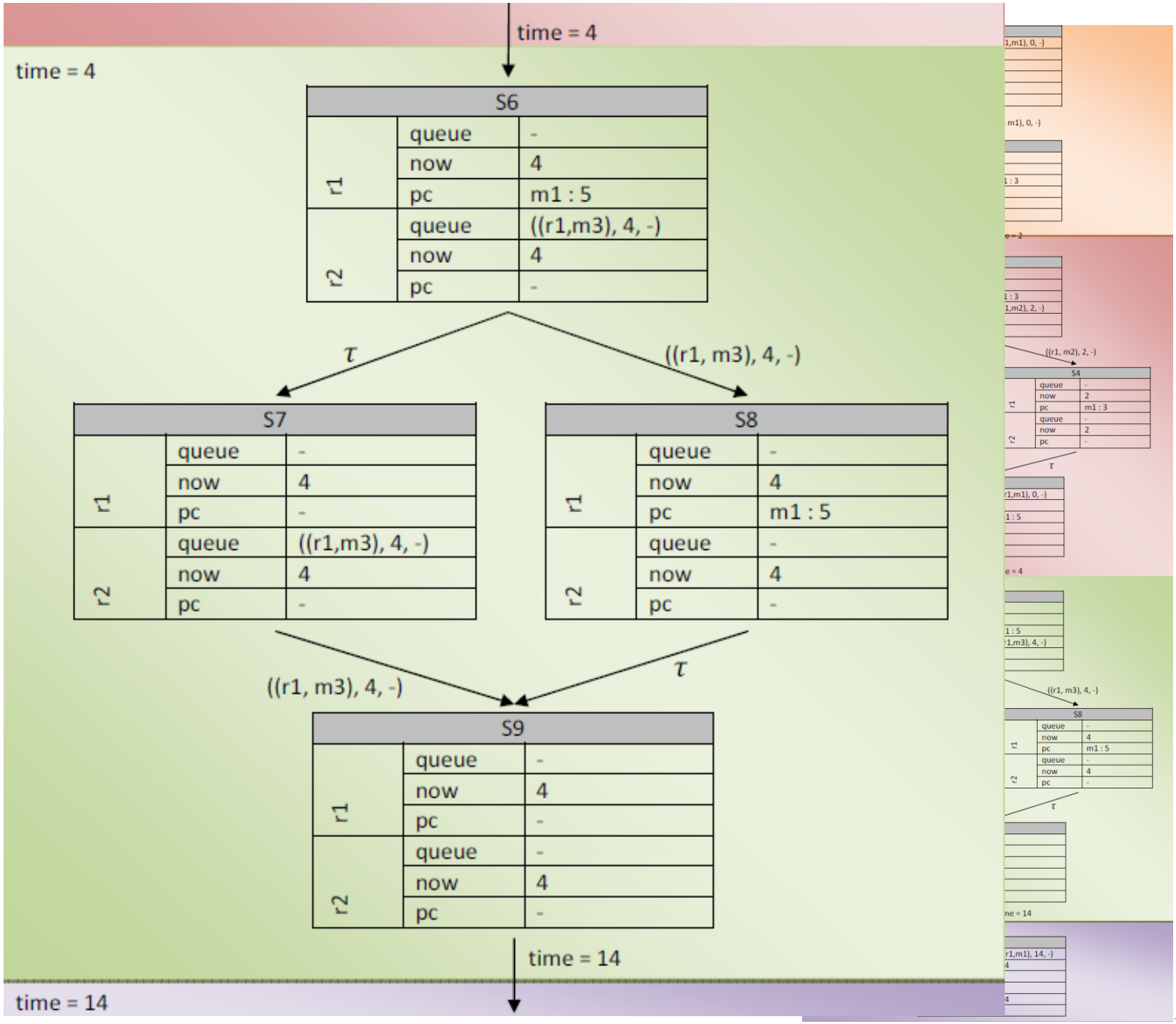




Bounde model ...nple

- Bounded system is
- Contents states are as FTTS



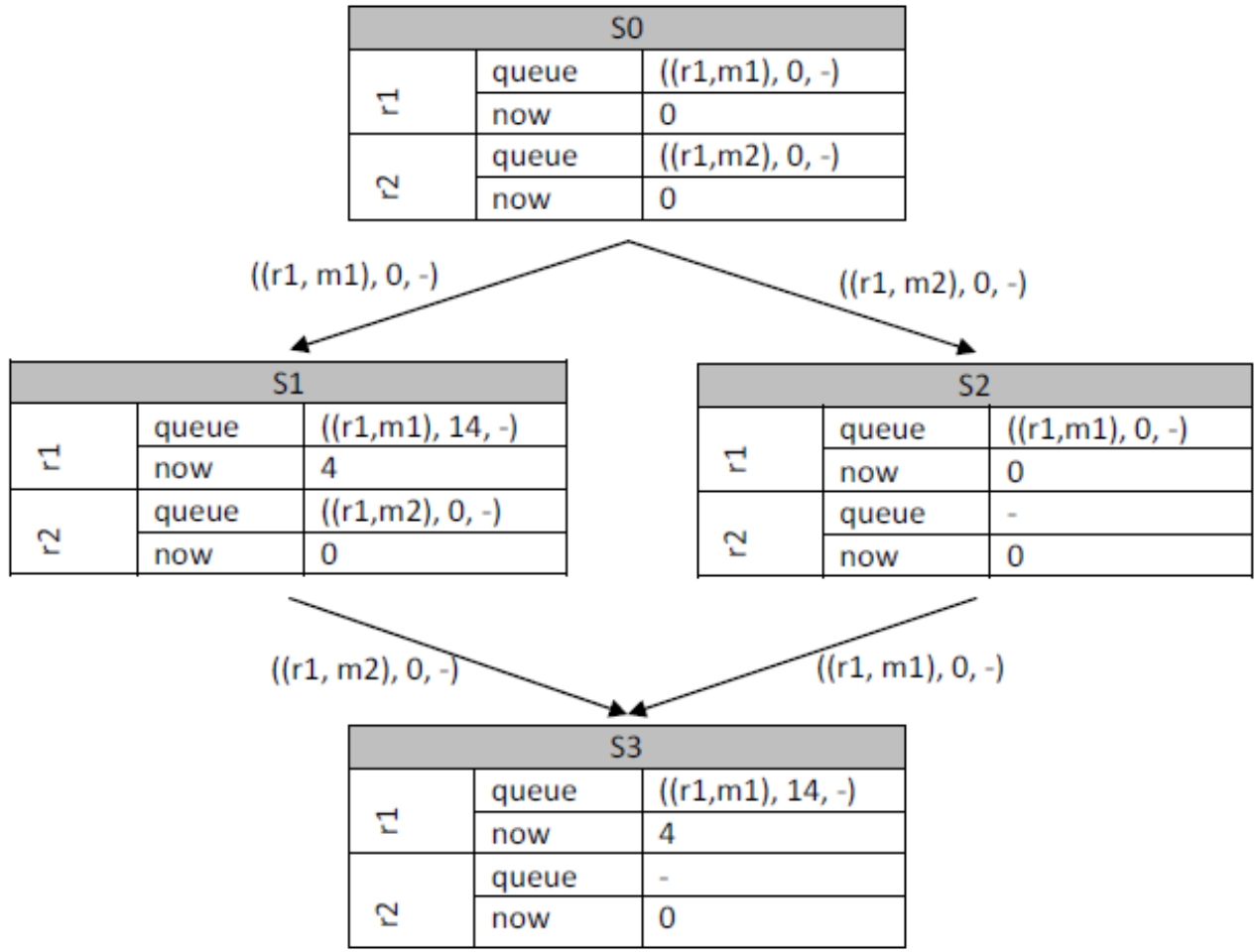

time = 4
time = 4
S6
r1
queue -
now 4
pc m1 : 5
r2
queue ((r1,m3), 4, -)
now 4
pc -
τ
((r1, m3), 4, -)
S7
r1
queue -
now 4
pc -
r2
queue ((r1,m3), 4, -)
now 4
pc -
S8
r1
queue -
now 4
pc m1 : 5
r2
queue -
now 4
pc -
((r1, m3), 4, -)
τ
S9
r1
queue -
now 4
pc -
r2
queue -
now 4
pc -
time = 14
time = 14

More Details about Floating-Time Transition System

- Floating-Time transition system may have

nc
re

reactive
(3) {
know
RC
}
RC1(
sel



Implementation and experimental results

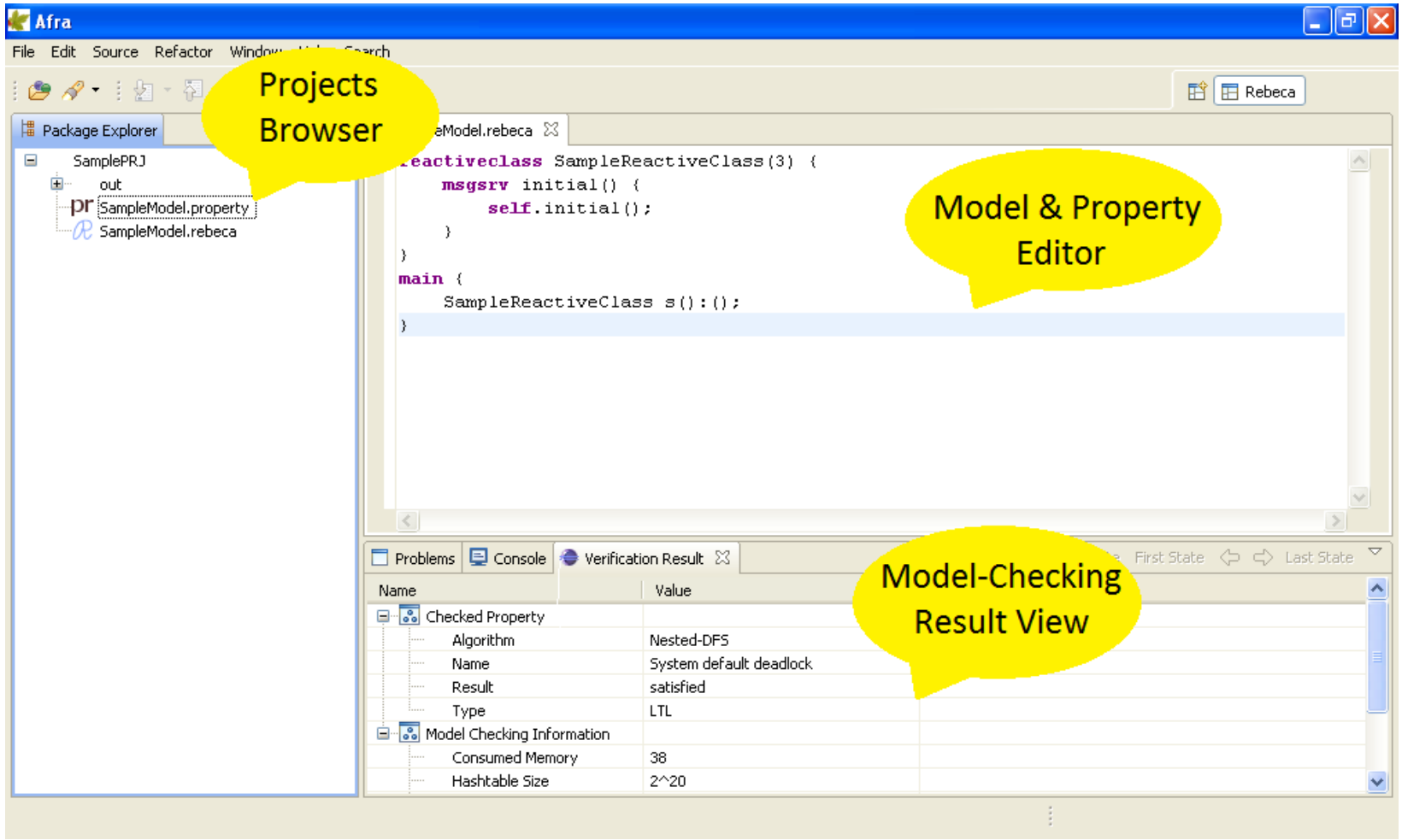
- Implemented over Rebeca model-checking platform
- Explores transition system by Breadth First Search (BFS) algorithm
- Added time bundle to the states to include time specifiers
  - There is tiny overhead because of time bundles
- Embedded in Afra tool set

Implementation and experimental results

- Three different models have been developed

Problem	Size	Using FTTS		Using Timed Automata	
		#states	time	#states	time
Ticket Service	**1 customer**	8	<1(sec)	801	<1(sec)
	2 customers	56	<1(sec)	19263811	≈ 5(hours)
	3 customers	20617	3(secs)	-	-
Sensor Net.	**1 sensor**	52	<1(sec)	-	-
	2 sensors	592	<1(sec)	-	-
	3 sensors	8154	1	-	-
	4 sensors	122900	18	-	-
Slotted ALOHA Protocol	**1 interface**	159	<1(sec)	-	-
	2 interfaces	2030	1(sec)	-	-
	3 interfaces	17253	5(secs)	-	-
	4 interfaces	132200	52(secs)	-	-
	5 interfaces	966147	490(secs)	-	-

Afra tool


Afra
File Edit Source Refactor Window
Rebeca
Package Explorer
SamplePRJ
out
SampleModel.property
SampleModel.rebeca
Projects Browser
eModel.rebeca
reactiveclass SampleReactiveClass(3) {
msgsrv initial() {
self.initial();
}
}
main {
SampleReactiveClass s():();
}
Model & Property Editor
Problems
Console
Verification Result
First State
Last State
Name
Value
Checked Property
Algorithm
Nested-DFS
Name
System default deadlock
Result
satisfied
Type
LTL
Model Checking Information
Consumed Memory
38
Hashtable Size
2^20
Model-Checking Result View

Comparing to others

- Real-time Maude
  - They have to tick – so, explosion
  - No wrap-around for the time – bounded model checking
- Timed Automata
  - Come up with many automata and many clocks for an asynchronous system - explode

Conclusion

- An actor-based modeling language for modeling real-time concurrent event-based systems
- Schedulability and deadlock freedom analysis
- State-space reduction techniques using the specific semantics

Our reduction technique: distilled

- Event-based analysis - maximum progress of time based on events (not timer ticks)
  - Generating no new states because of delays, each rebec has its own local time in each state
- Making use of isolated message server execution of actors
  - no shared variables, no blocking send or receive, single-threaded actors, non-preemptive execution of each message server
- A new notion of states equivalence by shifting the local times of concurrent elements in case of recurrent behaviors

Future work

- A lot!

Timed Event-based Property Language: TeProp

- Event-based rather than state-based
- Time intervals

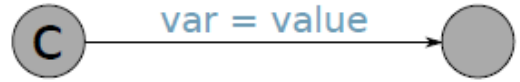
- We chose / designed our operators based on the timed patterns
  - Trying to have simpler properties for each pattern
  - Close to MTL

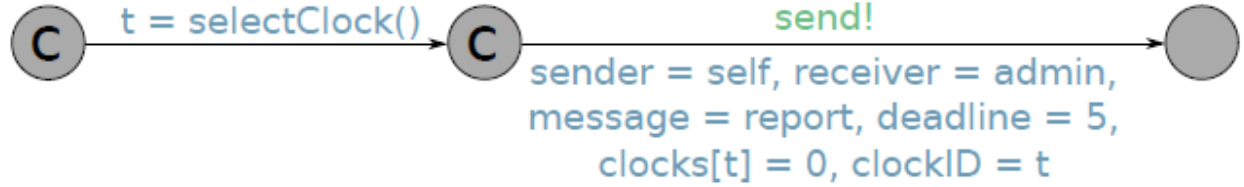
Timed-Rebeca to Timed Automata

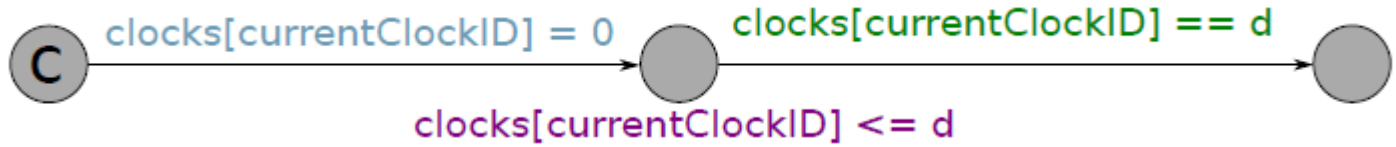
- There is no straightforward and efficient mapping from Timed-Rebeca to timed automata
  - There is asynchronous message passing in Timed-Rebeca, whereas timed automata communication is synchronous
  - Local times of each rebec needs to be modeled as a set of clocks in timed automata
  - Each undelivered message (messages with after) has a clock to model its late delivery
  - We need three types of automata for each rebec
    - One for *message servers (now)* (one clock)
    - A set for *after* (as many clocks as undelivered

Timed Automata for each Statement

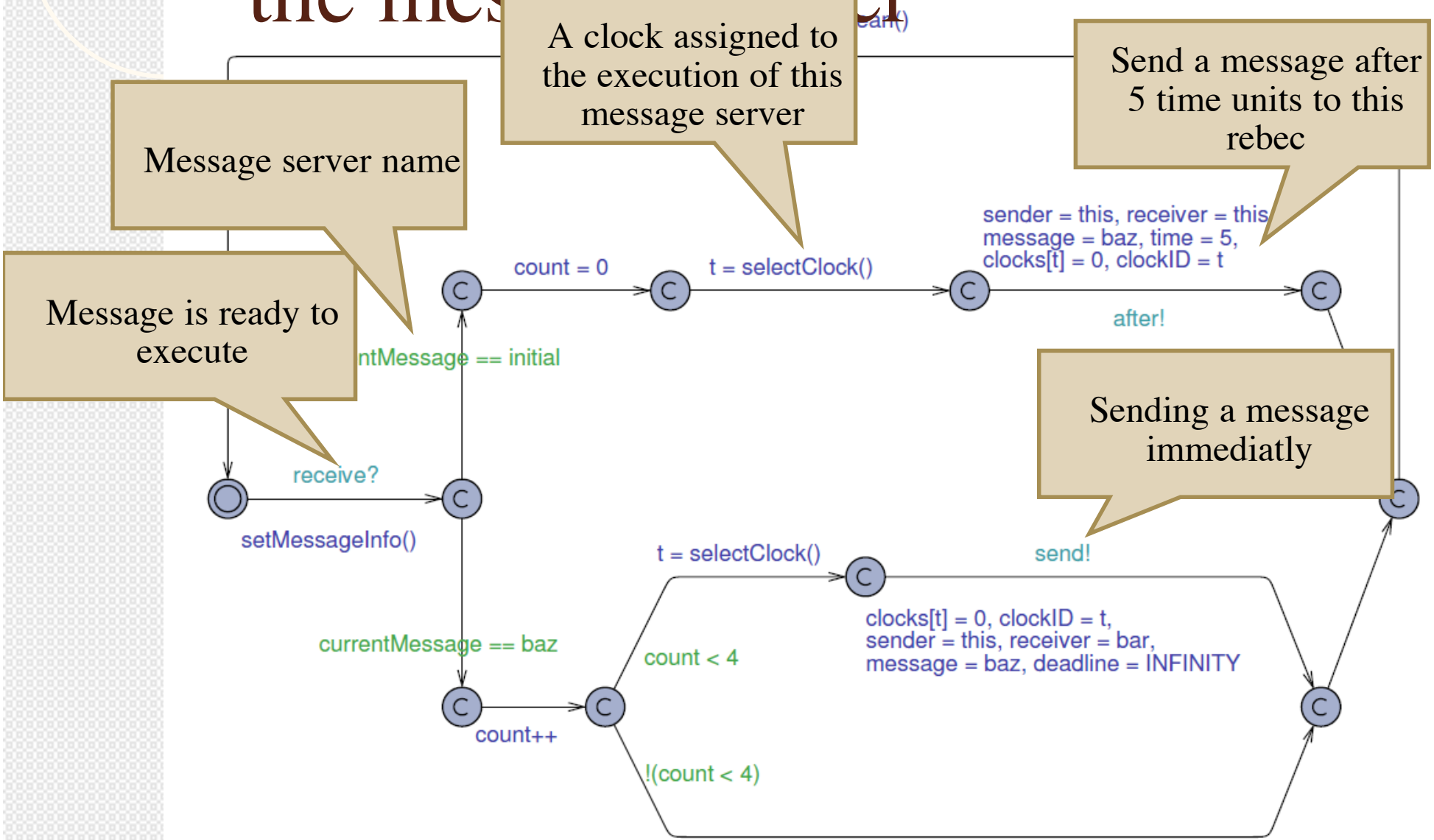
- All the message servers of a rebec can be modeled by a timed automaton
  - Assignment
  - Sending message
  - Delay





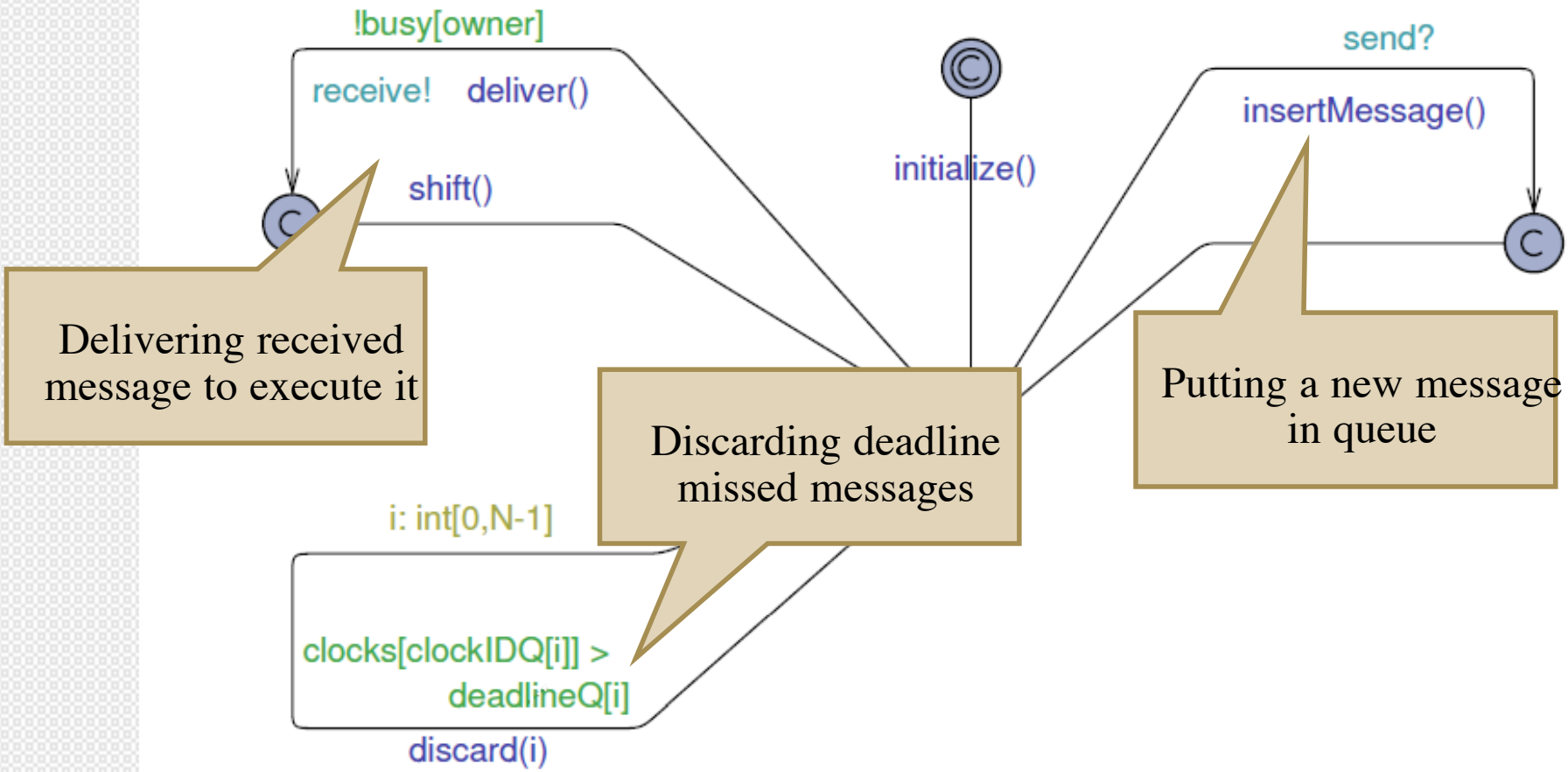


Timed Automata for the Body of the message server



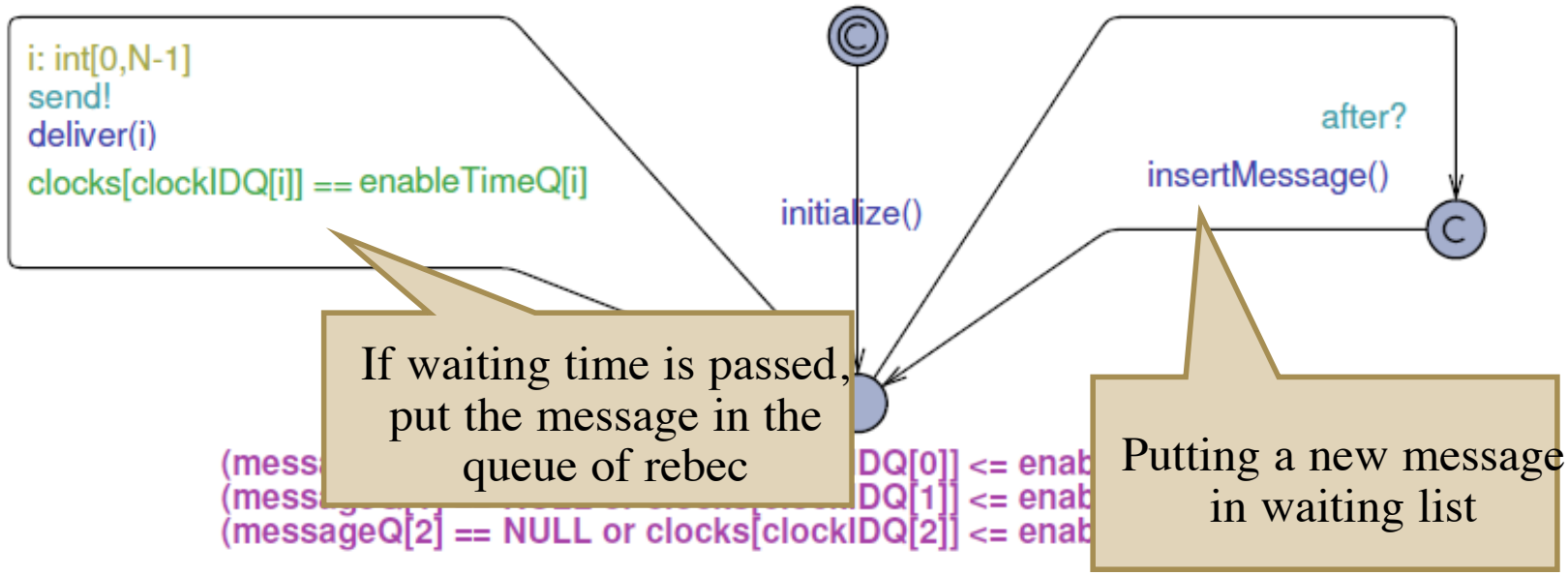
Timed Automata for the Queue

- Rebec message bag management can be modeled by a timed automaton



Timed Automata for *After*

- Messages which has “after” specifier can be modeled by a timed automaton



Timed-Rebeca to Timed Automata: summary

- Extremely inefficient in practice
  - Three (types of) automata per rebec
  - Clocks valuations make huge region transition system
  - After clocks: too much unnecessary interleaving and time regions