Schedulability, Deadlock Freedom, and Performance Analysis of Timed Actors

Marjan Sirjani

Reykjavik University, Iceland

DOP Center, EECS, UC Berkeley, June 2013

A joint work with

- Ramtin Khosravi
- Ehsan Khamespanah
- Brynjar Magnússon
- Haukur Kristinsson
- Ali Jafari
- Arni Herman Reynisson
- Steinar Hugi Sigurdarson
- Luca Aceto
- Anna Ingolfsdottir
- Matteo Cimini
- Zeynab Sabahi
- Zeynab Sharifi
- Mohammad Javad Izadi

From Reykjavik University and University of Tehran

Outline

- Motivation (and where do we stand)
- Rebeca and Timed Rebeca – the language
- Semantics and Floating Time Transition System
- Schedulability and Deadlock Freedom
- Simulation and Performance Analysis
- 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
- The best scheduling for multiple elevators in a building
- The best strategy for the European fishing market
- The most efficient rescue plan for a set of fire engines
- The best routing algorithm in a network

Analyzing Techniques

- Testing
- Simulation
- Analytical Techniques (performance analysis)
- Model checking (correctness)
- Theorem proving (correctness)

- Alternative approach:
  - State-based simulation and verification:

    Performance Evaluation and Model Checking Join Forces

    Baier, Haverkort, Hermanns, Katoen

    Communications of the ACM, 2010

A Good Model

A good model has to be:

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

Tradeoff!

Different approaches

Abstract

Mathematical

Modeling languages

CCS CSP
Petri net
RML
I/O Automata

FDR

Mocha

Verification Techniques:

- Deduction
  needs high expertise
- Model checking
  causes state explosion

SMV
Promela

NuSMV

Spin

Java PathFinder

Programming languages

Too heavy
Informal

Java
C

Bandera

SLAM

Our choice: Actors

- A reference model for concurrent computation
- Consisting of concurrent, distributed active objects

Proposed by Hewitt as an agent-based language (MIT, 1971)

Developed by Agha as a concurrent object-based language (UIUC, since 1984)

Formalized by Talcott (with Agha, Mason and Smith): Towards a Theory of Actor Computation (SRI, 1992)

Why actors?

- Usable: a nice language!
  - OO is familiar for practitioners
  - Simple and intuitive model of concurrency
- Analyzable: formal basis
  - we will provide a model checker
  - Loosely coupled actors will help in developing more efficient analysis techniques, like for compositional verification

So, ...

- We designed Rebeca language
- Developed model checking tools for it
- Established theories and tools for compositional verification and reduction techniques

Rebeca: The Modeling Language

- **<u>Re</u>**active o**<u>b</u>**<u>j</u>**<u>ec</u>**t l**<u>a</u>**nguage

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

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

(Hewitt-Agha Actors)

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
  - Event-driven
  - Non-preemptive (atomic execution)

Ticket Service model

```
reactiveclass TicketService {
  knownrebecs {
    Agent a;
  }
  statevars {
    int issueDelay;
  }
  msgsrv initial(int myDelay) {
    issueDelay = myDelay;
  }
  msgsrv requestTicket() {
    delay(issueDelay);
    a.ticketIssued(1);
  }
}
reactiveclass Agent {
  knownrebecs {
    TicketService ts;
    Customer c;
  }
  msgsrv requestTicket() {
    ts.requestTicket() deadline(5);
  }
```

Actor type and its message servers

```
  msgsrv ticketIssued(byte id) {
    c.ticketIssued(id);
  }
}
reactiveclass Customer {
  knownrebecs {
    Agent a;
  }
  msgsrv initial() {
    self.try();
  }
  msgsrv try() {
    a.requestTicket();
  }
  msgsrv ticketIssued(byte id) {
    self.try() after(30);
  }
}
main {
  Agent a(ts, c):();
  TicketService ts(a):(3);
  Customer c(a):();
}
```

Asynchronous message sending

Instances of three different actors

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.
- N. Razavi, R. Behjati, H. Sabouri, E. Khamespanah, A. Shali, M. Sirjani, **Sysfier: Actor-based Formal Verification of SystemC**, ACM Transactions on Embedded Computing Systems, Vol. 10, No. 2, Article 19, 2010.

Rebeca Formal Modeling Language

Rebeca

View Edit History Print

Home
Documentation
Projects
Tools
Publications
Members
Downloads
Examples

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.

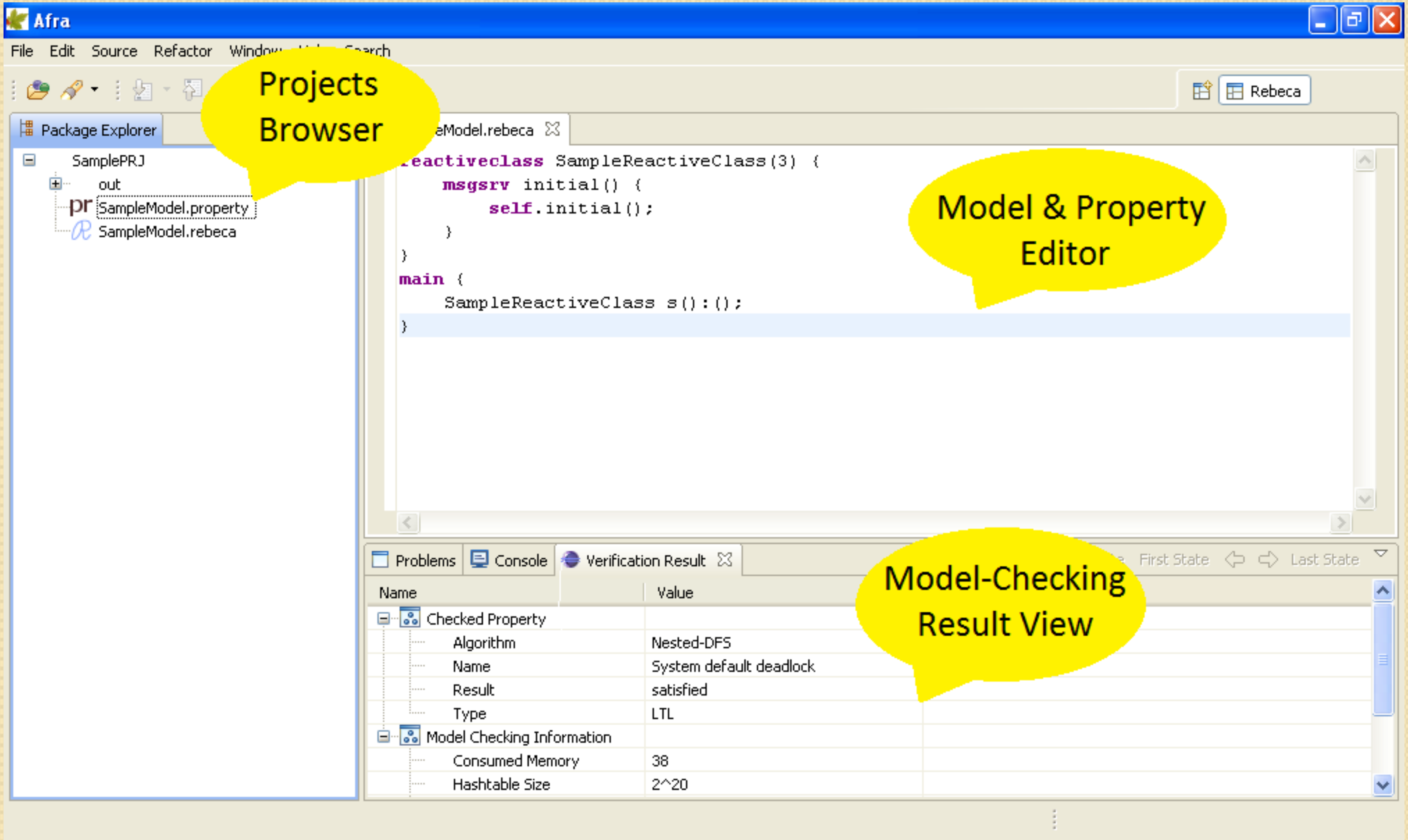
Besides having an appropriate and efficient way for modeling concurrent and distributed systems, one needs a formal verification approach to ensure their correctness. Rebeca is supported by Rebeca Verifier tool, as a front-end, to translate the codes into existing model-checker languages and thus, be 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.

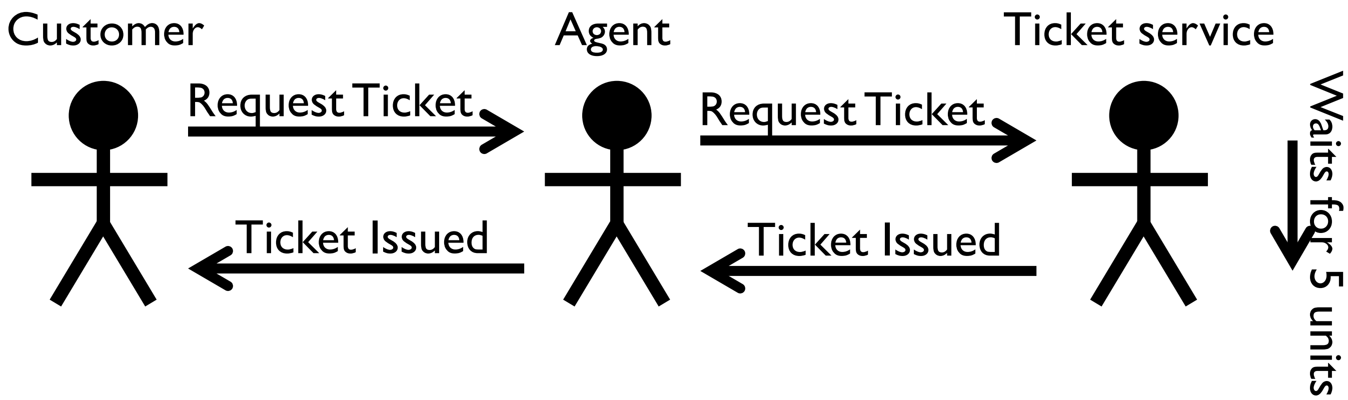

Afra
File Edit Source Refactor Window Search
Rebeca
Package Explorer
SamplePRJ
out
SampleModel.property
SampleModel.rebeca
eModel.rebeca
reactiveclass SampleReactiveClass(3) {
msgsrv initial() {
self.initial();
}
}
main {
SampleReactiveClass s():();
}
Projects Browser
Model & Property Editor
Model-Checking Result View
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

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 are time constraints 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);

...class Agent {
  ...rebecs {
    TicketService ts;
    Customer c;
  }
  msgsrv requestTicket() {
    ts.requestTicket() deadline(5);
  }
```

```
  msgsrv ticketIssued(byte id) {
    c.ticketIssued(id);
  }
}
reactiveclass Customer {
  knownrebecs {
    Agent a;
  }
  m... initial() {
    ...y();

  ...try() {
    ...uestTicket();
  }
  msgsrv ti...etIssued(byte id) {
    \\ customer happy
    self.try() after(30);

main {
  Agent a(ts, c):();
  TicketService ts(a):(...
  Customer c(a):();
}
```

Time progress because of computation delay

Acto... its ... s...

Communication delay or periodic tasks

Asynchronous message sending

Deadline for the message release

Instances of three different actors

Semantics of a simple Timed-Rebeca Model:

Timed Transition System

```
reactiveclass RC1 (3) {
    knownrebecs {
        RC2 r2;
    }
    ... {
        ...elf.m1();
    }
    msgsr...
        d...
        r2...
        d...
        r...
        se...
    }
}
```

```
reactiveclass RC2 (4) {
    knownrebecs {
        RC1 r1;
    }
    RC2() { }
    msgsrv m2() { }
    ... { }
    ...);
    ...);
```

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

msgsrv m1()
1 delay(2);
2 r2.m2();
3 delay(2);
4 r2.m3();
5 self.m1() af
}

time = 0

time = 2

time = 2

time = 4

time = 4

$time = time + 2$

S5		
r1	queue	-
	pc	$m1{:}4$
r2	queue	-
	pc	-

$\tau(r1)$

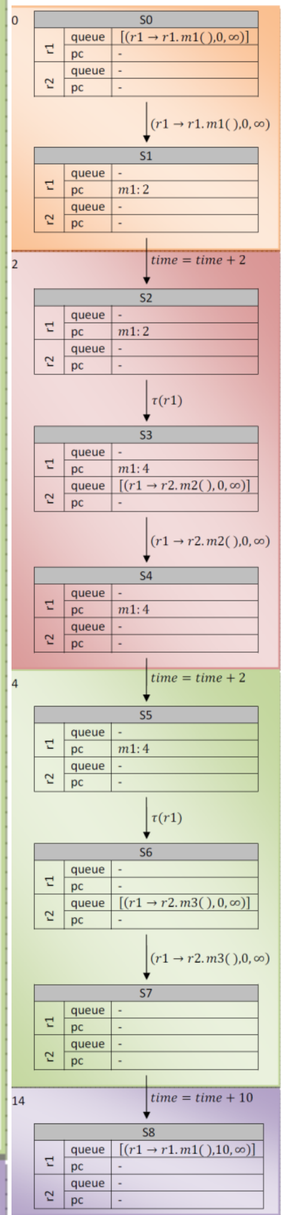
S6		
r1	queue	-
	pc	-
r2	queue	$[(r1 \rightarrow r2.m3(\,),0,\infty)]$
	pc	-

$(r1 \rightarrow r2.m3(\,),0,\infty)$

S7		
r1	queue	-
	pc	-
r2	queue	-
	pc	-

time = 14

$time = time + 10$



Transitions in TTS

- In TTS the transitions are of three types:
  - Passage of time
  - Taking a message from the queue to execute: event
  - Silent transition τ: internal actions in an actor

Properties in an event-based system

- Properties that we care about the most:
  - Distance of occurrence of two events
  - Event precedence

Real-time Patterns

(Koymans, 1990), (Abid et al., 2011), (Bellini et al., 2009) and (Konrad et al., 2005), (Dwyer et al., 1999)

- Maximal distance
  - Every e1 is followed by an *e*2 within *x* time units
- Exact distance
  - Every e1 is followed by an *e*2 in exactly *x* time units
- Minimal distance
  - Two consecutive events of *e* are at least *x* time units apart
- Periodicity
  - Event *e* occurs regularly with a period of *x* time units
- Bounded response
  - Each occurrence of an event *e* is responded within a maximum number of time units
- Precedence
  - Within the next *x* time units, the occurrence of *e*1 precedes the occurrence of *e*2

So, we proposed

- An event-based semantics for Timed Rebeca

Timed-Rebeca Semantics

- We introduced: Floating Time Transition System
- 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 smallest time tag of all the messages (for all the rebecs) in the message bag.
- The schedular checks if a ***deadline*** is missed.
- The variable ***now*** is set to the maximum between the current time of the rebec and the time tag of the selected message.

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:
	Message Bag: $[(null \rightarrow a.initial(\,),0,\infty)]$
	Now: 0
ts	State vars: issueDelay=?
	Message Bag: $[(null \rightarrow ts.initial(\,),0,\infty)]$
	Now: 0
c	State vars:
	Message Bag: $[(null \rightarrow c.initial(\,),0,\infty)]$
	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:	
	Message Bag: [ ]	
	Now:	3
ts	State vars:	issueDelay=3
	Message Bag: [ ]	
	Now:	3
c	State vars:	
	Message Bag: $[(a \rightarrow c.ticketIssued(1), 3, \infty)]$	
	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

s_{20} —Ticket Issued→ s_{21}

s_{21}		
a	State vars:	
	Message Bag: [ ]	
	Now:	36
ts	State vars:	issueDelay=3

$\cong_{33}$

s_{16}		
a	State vars:	
	Message Bag: [ ]	
	Now:	3
ts	State vars:	issueDelay=3

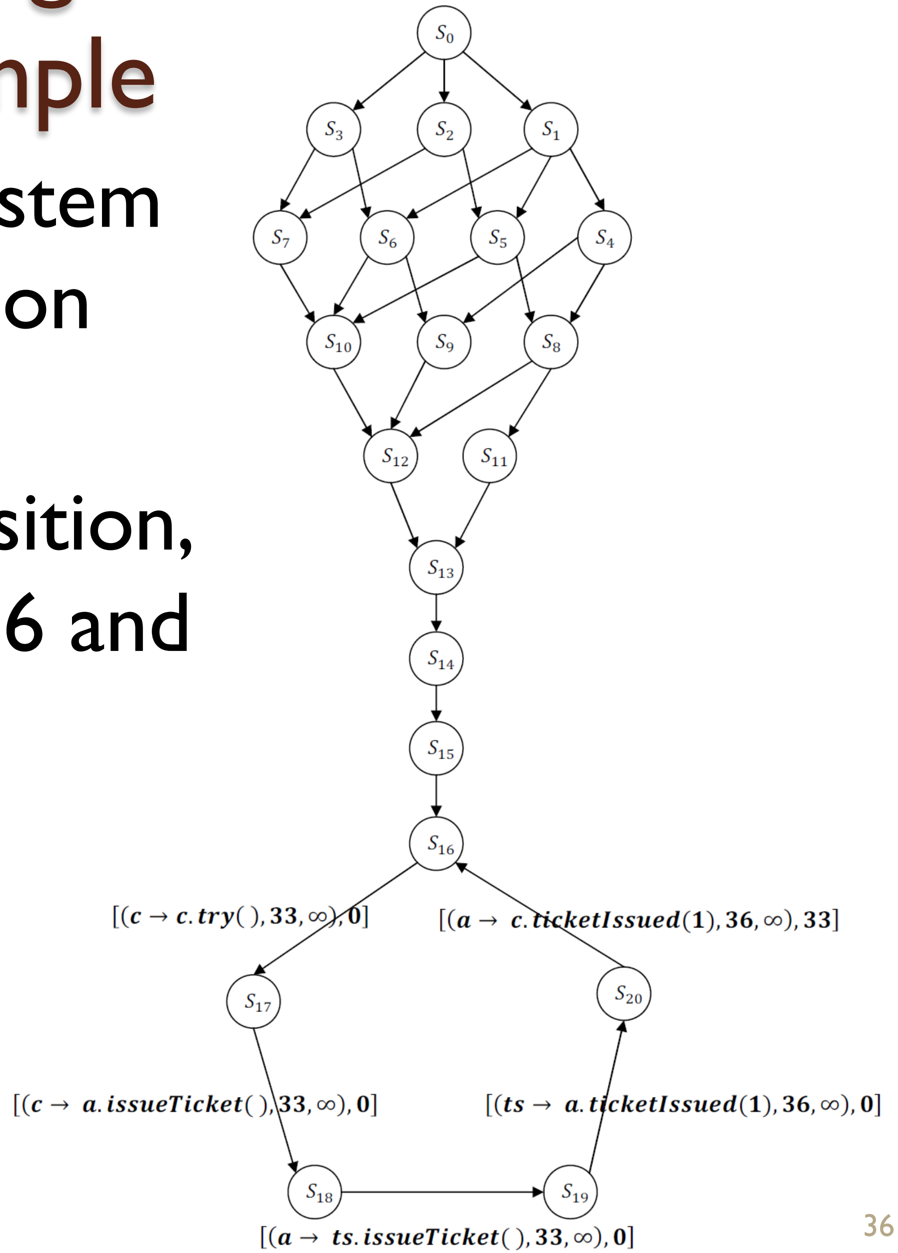
s_{20}		
a	State vars:	
	Message Bag: [ ]	
	Now:	36
ts	State vars:	issueDelay=3
	Message Bag: [ ]	
	Now:	36
c	State vars:	
	Message Bag: $[(a \rightarrow c.ticketIssued(1), 36, \infty)]$	
	Now:	3

Ticket Issued, 33 →

s_{16}		
a	State vars:	
	Message Bag: [ ]	
	Now:	3
ts	State vars:	issueDelay=3
	Message Bag: [ ]	
	Now:	3
c	State vars:	
	Message Bag: $[(c \rightarrow c.try(), 33, \infty)]$	
	Now:	3

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 execution of a message is the same in FTTS and Bounded FTSS
  - The message bag contents are the same
- Condition 2 holds because we are abstracting timing values

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-freedom" relative time is enough

State space reduction: a simple Timed-Rebeca Model

```
reactiveclass RC1 (3) {
    knownrebecs {
        RC2 r2;
    }
        {
        elf.m1();
    }
    msgsrv
        d
        r2
        d
        r
        se
    }
}
```

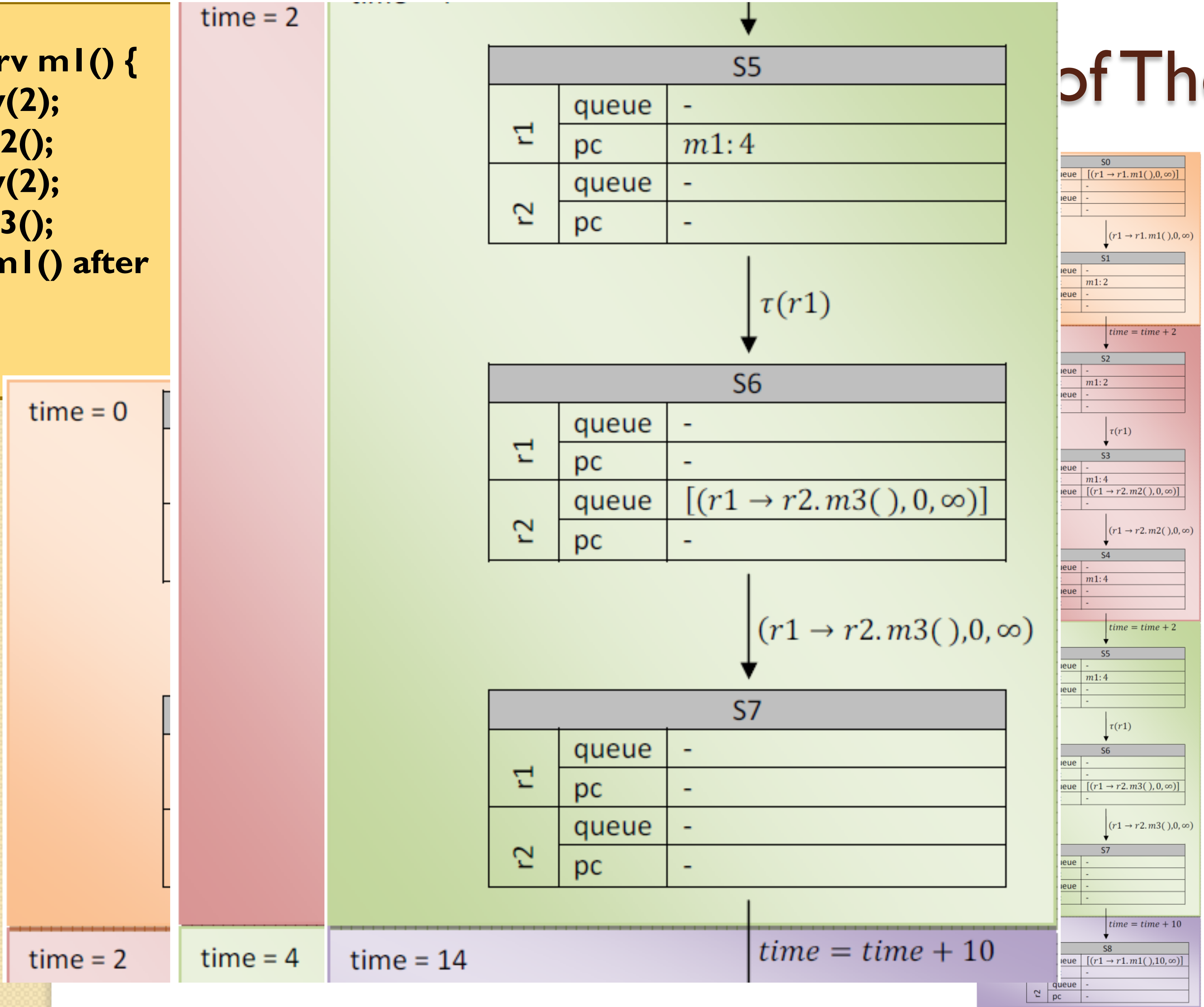
```
reactiveclass RC2 (4) {
    knownrebecs {
        RC1 r1;
    }
    RC2() { }
    msgsrv m2() { }
          { }
          );
          );
```

Line number as program counter

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

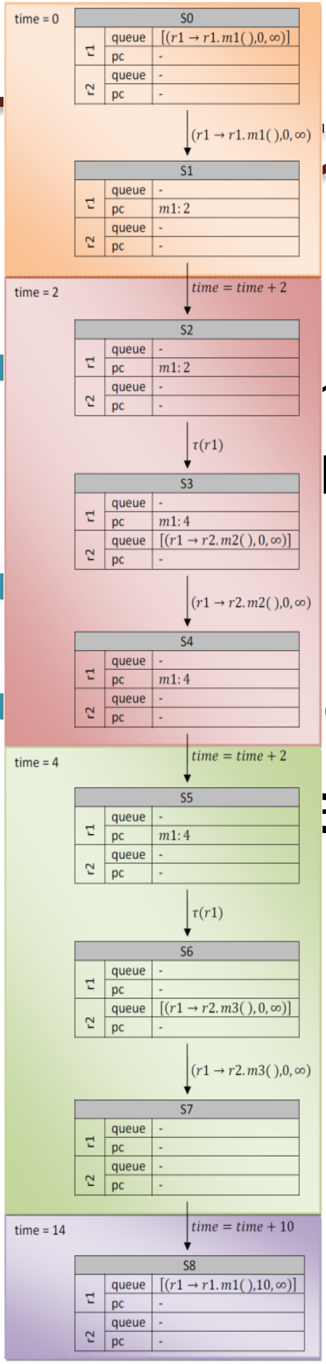
```
msgsrv m1() {
 delay(2);
 r2.m2();
 delay(2);
 r2.m3();
 self.m1() after
(10);
}
```

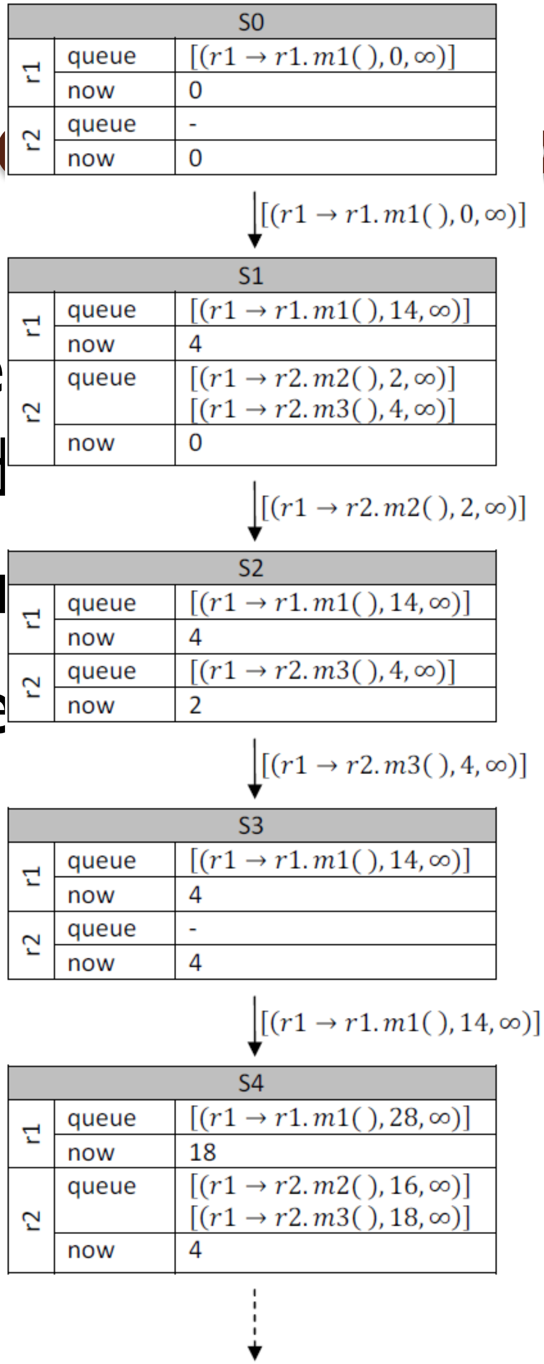
# of The

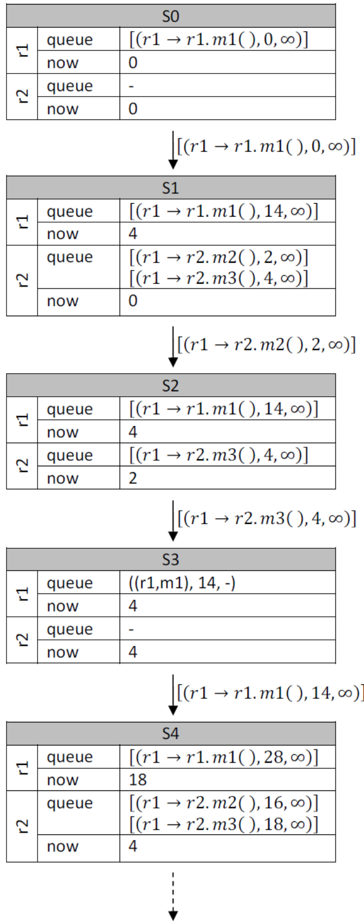


FT … odel

- … te … for
- … nd
- … o
- … de
- … ed

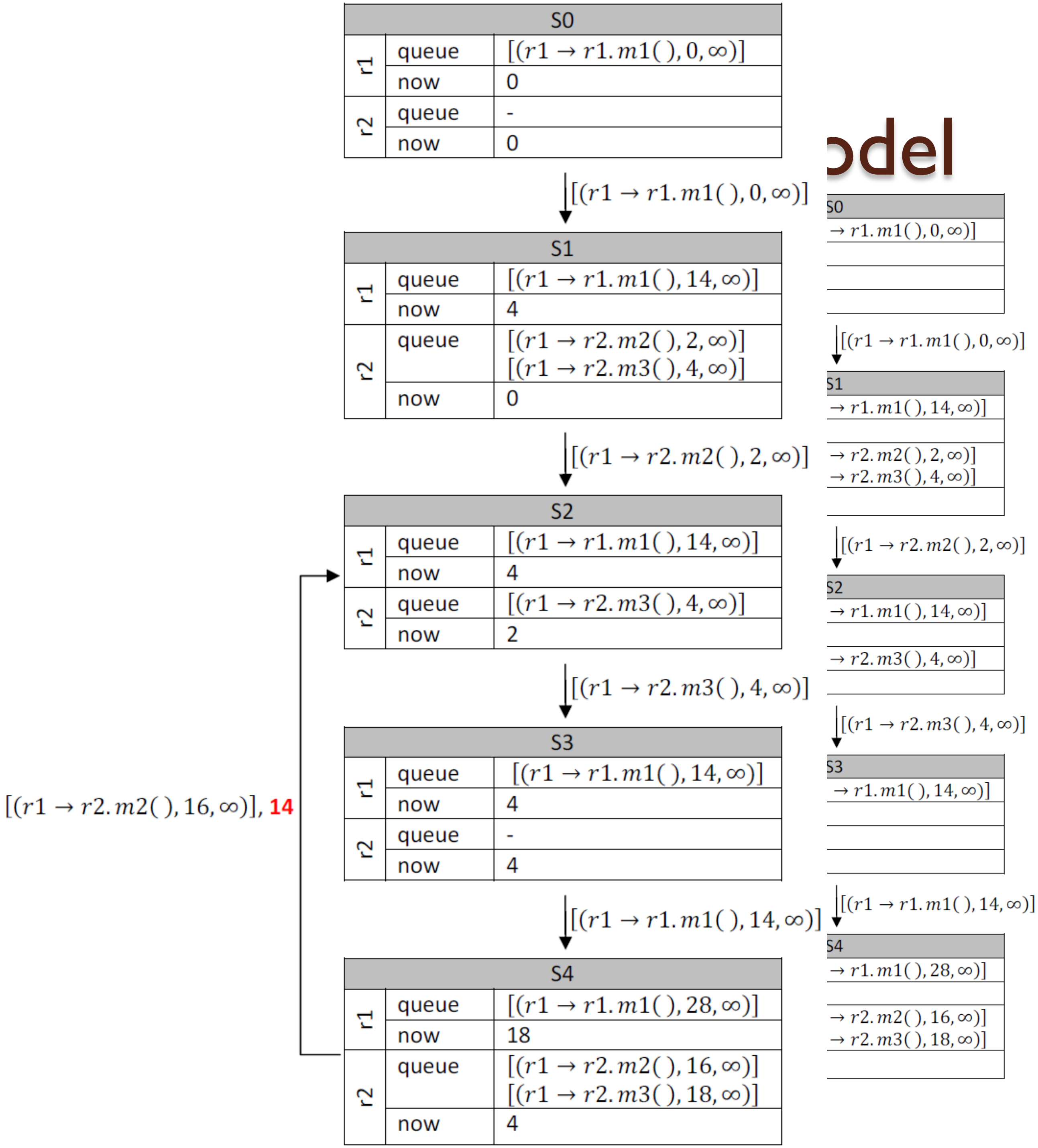






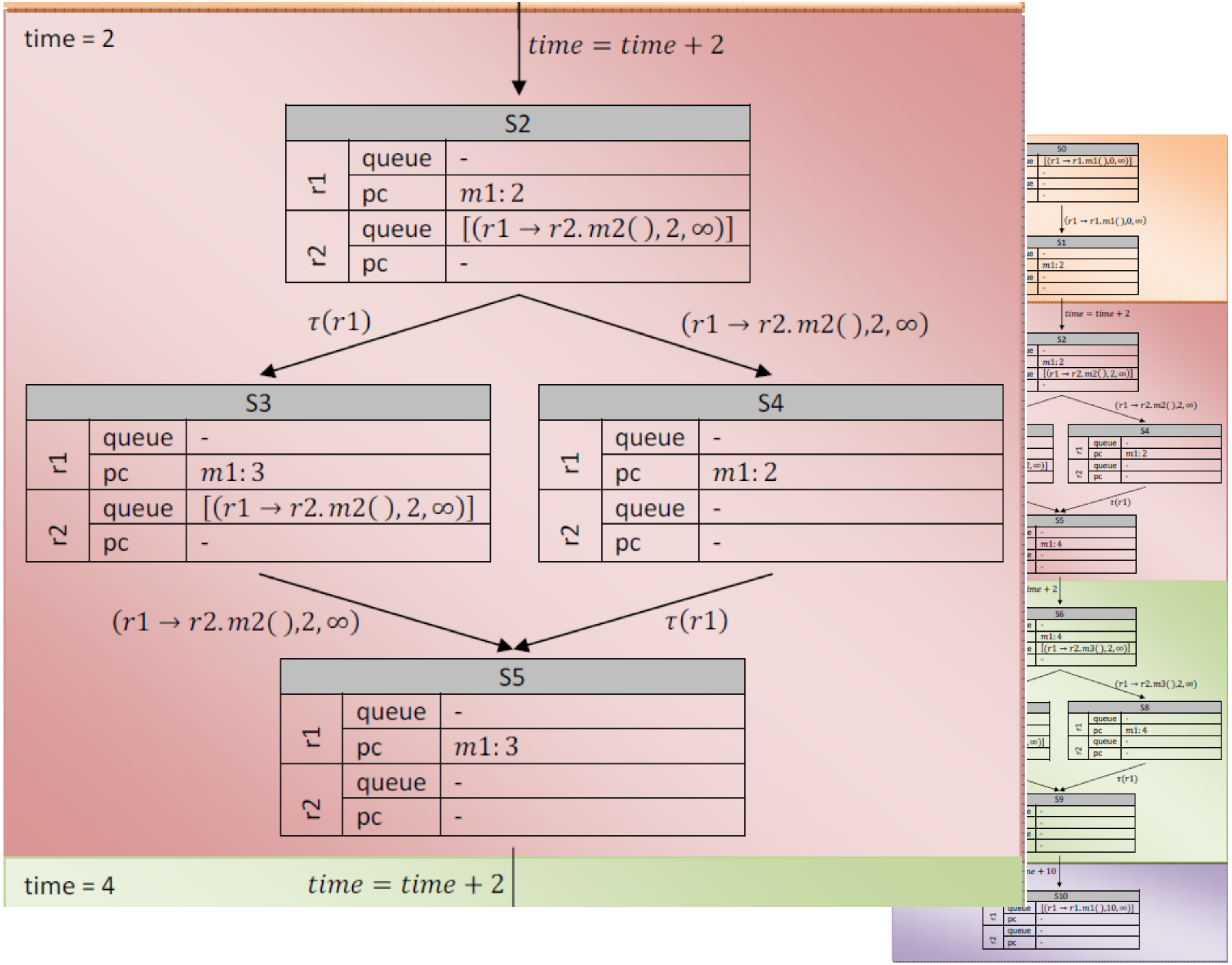
Bounde odel

- Bou syst
- Con state as F



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

Timed Transition System of a less simple model



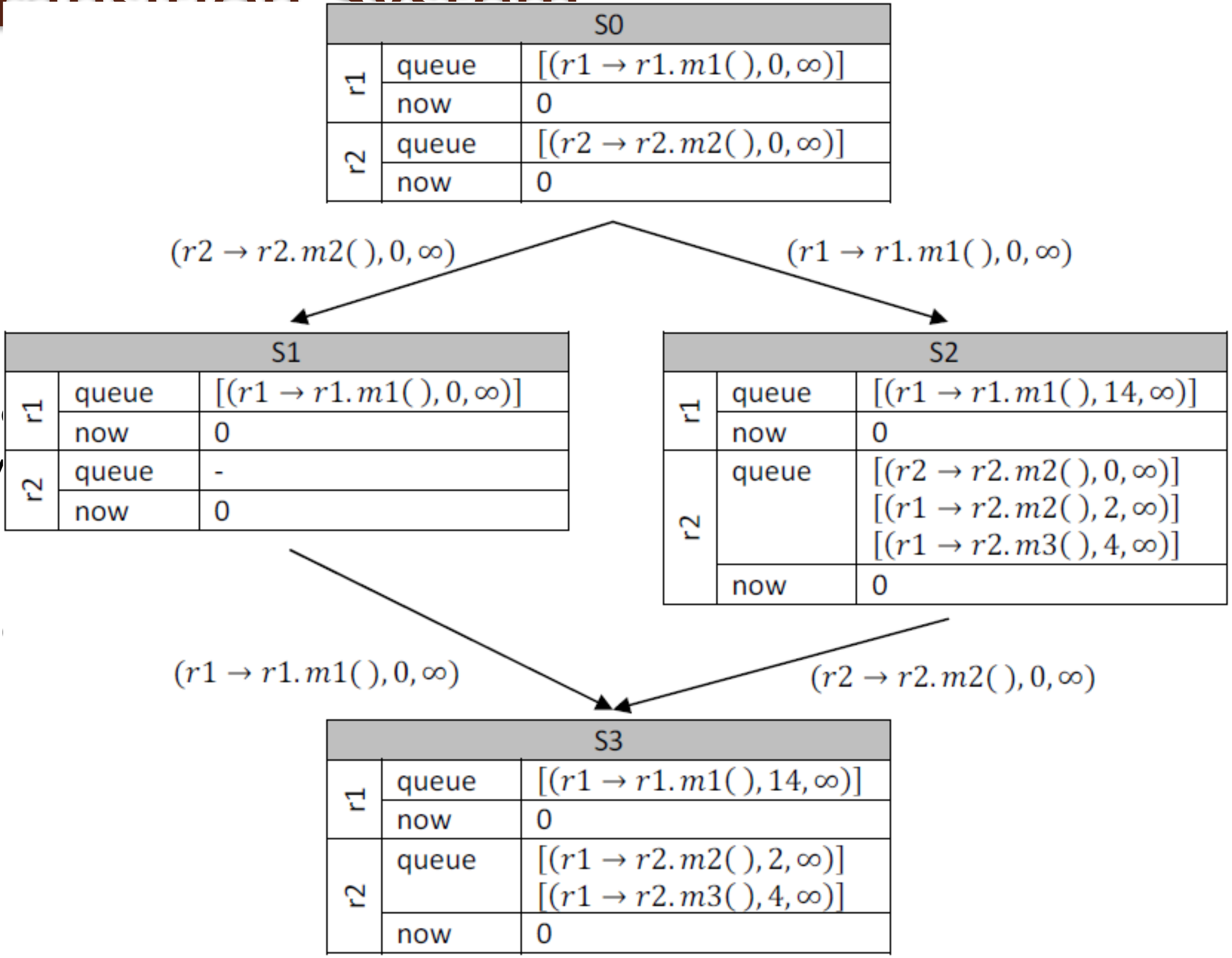
More Details about Floating-Time Transition System

-

reactive
know

}
RC1

}
…
}



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

TTS vs FTTS State Space Size

- About 50% state space reduction

Model Name	Number of Rebecs	FTTS State Space Size	TTS State Space Size
Ticket Service System	3	6	12
	4	43	86
	5	282	532
	6	2035	3526
	7	17849	31500
CSMA/CD	4	54	108

Implementation and experimental results

- Three different models have been developed

Problem	Size	Using BFTTS		Using Timed Automata		Using McErlang	
		#states	time	#states	time	#states	time
Ticket Service	1 customer	8	$<$ 1 sec	801	<1 sec	150	<1 sec
	2 customers	51	$<$ 1 sec	19M	5 hours	4.5k	3 secs
	3 customers	280	$<$ 1 sec	-	$>$24 hours†	190K	5.1 mins
	4 customers	1.63K	$<$ 1 sec	-	$>$24 hours†	$>$ 4M‡	-
	5 customers	11K	$<$ 1 sec	-	$>$24 hours†	$>$ 4M‡	-
	6 customers	83K	2 secs	-	$>$24 hours†	$>$ 4M‡	-
	7 customers	709K	3 mins	-	$>$24 hours†	$>$ 4M‡	-
	8 customers	6.8M	9.7 hours	-	$>$24 hours†	$>$ 4M‡	-
Sensor Network	1 sensor	183	$<$ 1 sec	-	$>$24 hours†	$>$ 6.5M‡	-
	2 sensors	2.4K	$<$ 1 sec	-	$>$24 hours†	$>$ 6M‡	-
	3 sensors	33.6K	1 sec	-	$>$24 hours†	$>$ 6M‡	-
	4 sensors	588K	13 secs	-	$>$24 hours†	$>$ 6M‡	-
Slotted ALOHA Protocol	1 interface	68	$<$ 1 sec	-	$>$24 hours†	153K	1.8 secs
	2 interfaces	750	$<$ 1 sec	-	$>$24 hours†	$>$ 2.8M‡	-
	3 interfaces	7.84K	1 sec	-	$>$24 hours†	$>$ 2.8M‡	-
	4 interfaces	45.7K	6 secs	-	$>$24 hours†	$>$ 2.8M‡	-
	5 interfaces	331K	64 secs	-	$>$24 hours†	$>$ 2.8M‡	-

Table 1: Model checking time and size of state space, using three different tools. The † sign on the reported time shows that model checking takes more than the time limit (24 hours). The ‡ sign on the reported number of states shows that state space explosion occurs as the model checker want to allocate more than 16GB in memory which is more than total amount of memory.

Implementation and experimental results

- Some minor changes in timing and behavior of models

Problem	Size	#states	time	result
Ticket Service	1 customer	5	< 1 sec	deadlock
	2 customers	25	< 1 sec	deadlock
	3 customers	180	< 1 sec	deadlock
	4 customers	1.4K	< 1 sec	deadlock
	5 customers	11.7K	< 1 sec	deadlock
	6 customers	108K	2 secs	deadlock
	7 customers	1.14	22 secs	deadlock
	8 customers	13M	7.6 mins	deadlock
Sensor Network	1 sensor	19	< 1 sec	deadlock
	2 sensors	147K	< 1 sec	deadlock
	3 sensors	23.7k	< 1 sec	deadlock
	4 sensors	1.14M	26 secs	deadlock
Slotted ALOHA Protocol	1 interface	57	< 1 sec	deadlock
	2 interfaces	277	< 1 sec	deadlock
	3 interfaces	1.2K	1 sec	deadlock
	4 interfaces	4.9K	1 sec	deadlock
	5 interfaces	20K	9 secs	deadlock

T: Table 3: Model checking the modified version of the three case studies which have deadlock state. n some cases.

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

Real-time Patterns

(Koymans, 1990), (Abid et al., 2011), (Bellini et al., 2009) and (Konrad et al., 2005), (Dwyer et al., 1999)

- Maximal distance
  - Every e1 is followed by an *e*2 within *x* time units
- Exact distance
  - Every e1 is followed by an *e*2 in exactly *x* time units
- Minimal distance
  - Two consecutive events of *e* are at least *x* time units apart
- Periodicity
  - Event *e* occurs regularly with a period of *x* time units
- Bounded response
  - Each occurrence of an event *e* is responded within a maximum number of time units
- Precedence
  - Within the next *x* time units, the occurrence of *e*1 precedes the occurrence of *e*2

We looked into Temporal Logic

- Metric Temporal Logic (MTL), real-time extension of Linear Temporal Logic (LTL)
- Timed Computational Tree Logic (TCTL), real-time extension of computational tree logic
- TILCO, logic language used for both specification and verification

P ::=

$\neg$P | P $\wedge$ P | P $\vee$ P | (P) |

F [i,j] e

F [i,j] (e $\rightarrow$P)

G [i,j] (e $\rightarrow$P)

e1 B[i,j] e2

Time interval: [i,j]

Interesting point

- For model checking TCTL or TLTL we need the complete timed transition system
- But we can check TeProp formulas on our BFTTS

- Two independent works
- In both the focus is on events and not the states

Outline

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

Simulation

- Simulation gives us more space for maneuver
- No state space explosion
- Trade off: no certain answer
- Good for prediction of performance

Simulation

- We translate Timed Rebeca models to Erlang.
  - Erlang is a functional programming language for programming real-time distributed systems.
  - Actor-based
- Use McErlang for simulation (and also model checking)

Using McErlang

- Variety of analysis techniques.
  - Verification
  - Visualization of simulations
  - Be able to model larger set of behaviors: extended Timed Rebeca
  - More variety of data-types.
  - Calling custom functions
- Add tracing capability to models to use McErlang
  - Checkpoints
  - Events

McErlang Safety monitors

- Acts like a safety property.
- Can observe each generated program state by McErlang.
  - on-the-fly verification.
  - violates or satisfies in each state
- Can access information from program states:
  - process mailboxes
  - status of the processes
  - process actions (sending or receiving)
- Pre-defined monitors
  - Deadlock detection
  - queue size of each process

Tracability of Simulations

- Events
  - Message send time
  - Message release time
  - Message expiration
- Checkpoints
  - Check point label and terms: values of variables
  - Time of checkpoint

Case studies on hand

- Routing in a Network-on-Chip
  - A joint work with the Hardware Department at Univ. of Tehran
  - Model checking and performance prediction

- Modeling the sensing application in TinyOS
  - Check the deadline miss

Probabilistic Timed Rebeca

- Model checking and performance analysis of timed probabilistic Rebeca based on PTA (and PFTTS?)

Back to Timed Rebeca and concluding …

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

Comparing to others

- Real-time Maude
  - They have to tick – so, explosion
  - 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
- Develop model checking and simulation tools
- Schedulability and deadlock freedom analysis
- State-space reduction techniques using the specific semantics

Future work

- A lot!