Schedulability and Compatibility of Real Time Concurrent Objects

Mohammad Mahdi Jaghoori, Delphine Longuet, Frank S. de Boer and Tom Chothia

CWI, Amsterdam

Real-Time Systems Symposium
1 December, 2008

This work was supported by the European project

http://cwi.nl
Motivation

- **Concurrent Objects**
  - Dedicated processors (for modeling distributed systems)
  - Asynchronous message passing
  - A process is created for handling each incoming message
    - There is a method corresponding to each message
  - Non-preemptive scheduling

- **Task Automata**
  - Extending Timed Automata with timed events
  - for specifying the generation patterns of processes
  - as non-uniformly recurring tasks

**Objective**

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
Motivation

- **Concurrent Objects**
  - Dedicated processors (for modeling distributed systems)
  - Asynchronous message passing
  - A process is created for handling each incoming message
    - There is a method corresponding to each message
  - Non-preemptive scheduling

- **Task Automata**
  - Extending Timed Automata with timed events
  - for specifying the generation patterns of processes
  - as non-uniformly recurring tasks

**Objective**

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
Motivation

- Concurrent Objects
  - Dedicated processors (for modeling distributed systems)
  - Asynchronous message passing
    - A process is created for handling each incoming message
      - There is a method corresponding to each message
  - Non-preemptive scheduling

- Task Automata
  - Extending Timed Automata with timed events
    - for specifying the generation patterns of processes
    - as non-uniformly recurring tasks

Objective

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
Motivation

- Concurrent Objects
  - Dedicated processors (for modeling distributed systems)
  - Asynchronous message passing
  - A process is created for handling each incoming message
    - There is a method corresponding to each message
  - Non-preemptive scheduling

- Task Automata
  - Extending Timed Automata with timed events
  - for specifying the generation patterns of processes
  - as non-uniformly recurring tasks

Objective

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
Motivation

- Concurrent Objects
  - Dedicated processors (for modeling distributed systems)
  - Asynchronous message passing
  - A process is created for handling each incoming message
    - There is a method corresponding to each message
  - Non-preemptive scheduling

- Task Automata
  - Extending Timed Automata with timed events
  - for specifying the generation patterns of processes
  - as non-uniformly recurring tasks

Objective

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
Motivation

- Concurrent Objects
  - Dedicated processors (for modeling distributed systems)
  - Asynchronous message passing
  - A process is created for handling each incoming message
    - There is a method corresponding to each message
  - Non-preemptive scheduling

- Task Automata
  - Extending Timed Automata with timed events
  - for specifying the generation patterns of processes
  - as non-uniformly recurring tasks

Objective

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
Motivation

- **Concurrent Objects**
  - Dedicated processors (for modeling distributed systems)
  - Asynchronous message passing
  - A process is created for handling each incoming message
    - There is a method corresponding to each message
  - Non-preemptive scheduling

- **Task Automata**
  - Extending Timed Automata with timed events
  - for specifying the generation patterns of processes
  - as non-uniformly recurring tasks

**Objective**

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
Motivation

- Concurrent Objects
  - Dedicated processors (for modeling distributed systems)
  - Asynchronous message passing
  - A process is created for handling each incoming message
    - There is a method corresponding to each message
  - Non-preemptive scheduling

- Task Automata
  - Extending Timed Automata with timed events
    - for specifying the generation patterns of processes
    - as non-uniformly recurring tasks

Objective

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
Motivation

- Concurrent Objects
  - Dedicated processors (for modeling distributed systems)
  - Asynchronous message passing
  - A process is created for handling each incoming message
    - There is a method corresponding to each message
  - Non-preemptive scheduling

- Task Automata
  - Extending Timed Automata with timed events
  - for specifying the generation patterns of processes
    - as non-uniformly recurring tasks

Objective

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
Motivation

- Concurrent Objects
  - Dedicated processors (for modeling distributed systems)
  - Asynchronous message passing
  - A process is created for handling each incoming message
    - There is a method corresponding to each message
  - Non-preemptive scheduling

- Task Automata
  - Extending Timed Automata with timed events
  - for specifying the generation patterns of processes
  - as non-uniformly recurring tasks

Objective

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
Motivation

- Concurrent Objects
  - Dedicated processors (for modeling distributed systems)
  - Asynchronous message passing
  - A process is created for handling each incoming message
    - There is a method corresponding to each message
  - Non-preemptive scheduling

- Task Automata
  - Extending Timed Automata with timed events
  - for specifying the generation patterns of processes
  - as non-uniformly recurring tasks

Objective

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
Motivation

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.

Objective

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
Motivation

Objective

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
Motivation

Objective

Using Task Automata for specifying the task generation pattern for real-time concurrent objects for modular schedulability analysis.
1. Modeling Real-Time Objects
   - A behavioral interface
   - A class implementing the interface
   - A scheduler

2. Analysis
   - Schedulability
   - Compatibility

3. Conclusion
Modeling in UPPAAL

- We use Timed Automata templates parameterized in
  - Object identity: \texttt{self}
  - Known-objects

- The IDs are basically used for sending messages
Modeling a Behavioral Interface

- Modeling the behavioral interface
  - What inputs;
  - What outputs;
  - their order and timings (= behavior); and,
  - deadlines for inputs (will be used as schedulability requirement)
Modeling a Behavioral Interface

- Modeling the behavioral interface
  - What inputs;
  - What outputs;
  - their order and timings (= behavior); and,
  - deadlines for inputs (will be used as schedulability requirement)
Modeling a Behavioral Interface

- Modeling the behavioral interface
  - What inputs;
  - What outputs;
    - their order and timings (= behavior); and,
    - deadlines for inputs (will be used as schedulability requirement)
Modeling a Behavioral Interface

- Modeling the behavioral interface
  - What inputs;
  - What outputs;
  - their order and timings (= behavior); and,
  - deadlines for inputs (will be used as schedulability requirement)
Modeling the behavioral interface

- What inputs;
- What outputs;
- their order and timings (= behavior); and,
- deadlines for inputs (will be used as schedulability requirement)
Modeling a Behavioral Interface

- Modeling the behavioral interface
  - What inputs;
  - What outputs;
  - their order and timings (= behavior); and,
  - deadlines for inputs (will be used as schedulability requirement)

Mutual Exclusion handler (MutEx)
Modeling a Behavioral Interface

- Sending/receiving messages by
  - \texttt{invoke[message][receiver][sender]}

Mutual Exclusion handler (MutEx)
A class is a set of methods.

- `initial` method is for initialization
- Sending messages on the `invoke` channel together with specifying a deadline.
- Controlling method execution
  - `start` channel: starts a method, which then runs till the end.
  - `finish` channel: method finished, so schedule the next one.
Modeling a Class

- A class is a set of methods.
  - *initial* method is for initialization
- Sending messages on the `invoke` channel
  - together with specifying a deadline.
- Controlling method execution
  - `start` channel: starts a method, which then runs till the end.
  - `finish` channel: method finished, so schedule the next one.

\[ x \leq 1 \Rightarrow \text{finish}[self] ! \]
\[ x = 0 \]
\[ x = 1 \Rightarrow \text{taken} = \text{false} \]
Modeling a Class

- A class is a set of methods.
  - `initial` method is for initialization
- Sending messages on the `invoke` channel
  - together with specifying a `deadline`
- Controlling method execution
  - `start` channel: starts a method, which then runs till the end.
  - `finish` channel: method finished, so schedule the next one.
Modeling a Class

- A class is a set of methods.
  - initial method is for initialization
- Sending messages on the invoke channel
  - together with specifying a deadline.

Controlling method execution
- start channel: starts a method, which then runs till the end.
- finish channel: method finished, so schedule the next one.
Modeling a Class

- A class is a set of methods.
  - initial method is for initialization
- Sending messages on the invoke channel
  - together with specifying a deadline.
- Controlling method execution
  - start channel: starts a method, which then runs till the end.
  - finish channel: method finished, so schedule the next one.
Modeling a Class

- A class is a set of methods.
  - *initial* method is for initialization
- Sending messages on the *invoke* channel
  - together with specifying a *deadline*.
- Controlling method execution
  - *start* channel: starts a method, which then runs till the end.
  - *finish* channel: method finished, so schedule the next one.
A class is a set of methods.

- **initial** method is for initialization

Sending messages on the **invoke** channel

- together with specifying a **deadline**.

Controlling method execution

- **start** channel: starts a method, which then runs till the end.
- **finish** channel: method finished, so schedule the next one.
Modeling a Class

- A class is a set of methods.
  - `initial` method is for initialization
- Sending messages on the `invoke` channel
  - together with specifying a `deadline`.
- Controlling method execution
  - `start` channel: starts a method, which then runs till the end.
  - `finish` channel: method finished, so schedule the next one.

```
x <= 1
start[initial][self] ? x = 0
x == 1
finish[self] !
taken = false
```

MutEx: initial
Modeling a Method

- An object has a single thread of execution.
- We assume that a method runs till the end.
- If a method cannot do its job now?
  - For example, to wait for a message.
  - A self-call for the rest of the job.
  - delegate channel: inherits the remaining deadline.
Modeling a Method

- An object has a single thread of execution.
- We assume that a method runs till the end.
- If a method cannot do its job now?
  - For example, to wait for a message.
  - a self-call for the rest of the job.
  - delegate channel: inherits the remaining deadline.
Modeling a Method

- An object has a single thread of execution.
- We assume that a method runs till the end.
- If a method cannot do its job now?
  - For example, to wait for a message.
  - a self-call for the rest of the job.
  - `delegate` channel: inherits the remaining deadline.
Modeling a Method

- An object has a single thread of execution.
- We assume that a method runs till the end.
- If a method cannot do its job now?
  - For example, to wait for a message.
  - a self-call for the rest of the job.
  - delegate channel: inherits the remaining deadline.
Modeling a Method

- An object has a single thread of execution.
- We assume that a method runs till the end.
- If a method cannot do its job now?
  - For example, to wait for a message.
  - a self-call for the rest of the job.
  - `delegate` channel: inherits the remaining deadline.
Modeling a Method

- An object has a single thread of execution.
- We assume that a method runs till the end.
- If a method cannot do its job now?
  - For example, to wait for a message.
  - a self-call for the rest of the job.
  - delegate channel: inherits the remaining deadline.
Modeling a Method

- An object has a single thread of execution.
- We assume that a method runs till the end.
- If a method cannot do its job now?
  - For example, to wait for a message.
  - a self-call for the rest of the job.
  - `delegate` channel: inherits the remaining deadline.

```latex
\begin{align*}
  \quad x & \leq 3 \\
  \quad x & \leq 2 \\
  \quad x & \leq 2 \\
  \quad x & \leq 1 \quad \text{start[reqL][self]} ? \\
  \quad x & = 0 \\
  \quad \text{finish[self]} ! \\
  \quad \text{finish[self]} ! \\
  \quad x & = 2 \quad \text{delegate[reqL][self]} ! \\
  \quad x & = 3 \quad \text{deadline} = 12 \\
  \quad x & = 2 \quad \text{invoke[permitL][Left][self]} ! \\
  \quad x & = 1 \quad \text{taken} = \text{true} \\
  \quad x & = 1 \quad \text{taken} = \text{false} \\
  \quad x & \leq 2 \\
  \quad x & \leq 3 \\
  \quad \text{start[reqL][self]} ? 
\end{align*}
```

MutEx: reqL
Modeling Schedulers

- Queue
- Strongly input-enabled
- Context-switch (dispatching methods)
- Error state
Modeling Schedulers

- **Queue**
- **Strongly input-enabled**
- **Context-switch (dispatching methods)**
- **Error state**

General Scheduler

```latex
\text{msg : int[0,MSG]}
\text{delegate[\text{msg}][\text{self}]?}
\text{insertDelegate(\text{msg})}

\text{msg : int[0,MSG],}
\text{sender : int [0,OBJ-1]}
\text{invoke[\text{msg}][\text{self}][\text{sender}]?}
\text{insertInvoke(\text{msg}, \text{sender})}

\text{i : int[0,MAX-1]}
\{guard on i for policy\}
\&\& \text{isEnabled(q[i])}
\text{finish [self] ?}
\text{run := i, shift()}

\text{tail == 1}
\text{finish [self] ?}
\text{shift()}

\text{start[q[run]][self] !}

\text{tail > MAX}
\text{counter[i] > 0 \&\&}
\text{x[i] > d[i]}
```
Schedulability Analysis

- Schedulable objects need a queue-length of at most $\left\lceil \frac{d_{max}}{b_{min}} \right\rceil$

- Schedulability = Is the Error state reachable?
  - check for deadlock
$S = \text{Method automata} + \text{Scheduler automata (including queues)}$
Compatibility Checking

\[ B = \text{Behavioral Interfaces} \]

\[ S = \text{Method automata} + \text{Scheduler automata (including queues)} \]
Compatibility Checking

- Compatibility is defined as: $S_{obs} \subseteq B$
- $B$ is deterministic, therefore, trace inclusion is decidable
- $S$ is very big, so we cannot verify it

Test compatibility

- Counter-examples to compatibility are reported.
Compatibility Checking

Compatibility is defined as: \( S_{obs} \subseteq B \)

- \( B \) is deterministic, therefore, trace inclusion is decidable
- \( S \) is very big, so we cannot verify it

Test compatibility
- Counter-examples to compatibility are reported.
Compatibility Checking

- Compatibility is defined as: $S_{obs} \subseteq B$
- $B$ is deterministic, therefore, trace inclusion is decidable
- $S$ is very big, so we cannot verify it

- Test compatibility
  - Counter-examples to compatibility are reported.
Compatibility Checking

- Compatibility is defined as: $S_{obs} \subseteq B$
- $B$ is deterministic, therefore, trace inclusion is decidable
- $S$ is very big, so we cannot verify it

- Test compatibility
  - Counter-examples to compatibility are reported.
Compatibility is defined as: $S_{obs} \subseteq B$

$B$ is deterministic, therefore, trace inclusion is decidable

$S$ is very big, so we cannot verify it

Test compatibility

- **Counter-examples** to compatibility are reported.
Testing Compatibility

\[ B = \]
Behavioral Interfaces

\[ S = \]
Method automata
+ Scheduler automata (including queues)
Testing Compatibility

\[ B = \]
Behavioral Interfaces

\[ S = \]
Method automata
+ Scheduler automata
(including queues)
Testing Compatibility

\[ B = \]
Behavioral Interfaces

\[ S = \]
Method automata
+ Scheduler automata (including queues)
Testing Compatibility

\[ B = \]
Behavioral Interfaces

\[ S = \]
Method automata
+ Scheduler automata (including queues)
Conclusions and Future Work

- We adapt task automata for specifying behavioral interfaces
  - Considering self-calls/delegation
  - Tasks specified (instead of best and worst-case execution times)

- Schedulability verified for each object
- Compatibility is tested for a composition of objects

Further work
- Schedulability analysis for complete Creol language
  - Synchronous communication
  - Processor release points
- Scheduler specification
- Multi-core processors (mutli-threaded objects)
Questions

email: jaghouri@cwi.nl