Schedulability Analysis of Concurrent Objects

Mohammad Mahdi Jaghoori

Centrum voor Wiskunde en Informatica
CWI, Amsterdam

IPA Spring days
8 May 2008
Task Automata - Example

A(10)!
\(x > 7\)
\(x := 0\)

B(6)!
\(x > 5\)
\(x := 0\)

A:
best-case = 3
worst-case = 4

B:
best-case = 2
worst-case = 5
Task Automata - Example

A:
- best-case = 3
- worst-case = 4

B:
- best-case = 2
- worst-case = 5

A(10)!
\(x > 7\)
\(x := 0\)

B(6)!
\(x > 5\)
\(x := 0\)
Task Automata - Example

A(10)!
\[x > 7\]
\[x := 0\]

\[x > 5\]
\[B (6)!\]
\[x := 0\]

A:
best-case = 3
worst-case = 4

B:
best-case = 2
worst-case = 5

A:
\[\text{best-case} = 3\]
\[\text{worst-case} = 4\]

B:
\[\text{best-case} = 2\]
\[\text{worst-case} = 5\]
Task Automata - Example

A(10)!
\[x > 7\]
\[x := 0\]

B(6)!
\[x > 5\]
\[x := 0\]

A:
- best-case = 3
- worst-case = 4

B:
- best-case = 2
- worst-case = 5
Task Automata - Example

A(10) !
x > 7
x := 0

B (6) !
x > 5
x := 0

A :
best-case = 3
worst-case = 4

B :
best-case = 2
worst-case = 5
Task automata

- Task automata model real time systems with
  - Non-uniformly recurring tasks
  - Single processor

- Timed automata extended with dynamically created tasks

- Task characteristics
  - an abstraction of the computation (best-case and worst-case execution time)
  - generated by timed events (timing constraints in timed automata)
  - must finish within its (relative) deadline
  - its completion time may influence release of other tasks
Task automata

- Task automata model real time systems with
  - Non-uniformly recurring tasks
  - Single processor

- Timed automata extended with dynamically created tasks

- Task characteristics
  - an abstraction of the computation (best-case and worst-case execution time)
  - generated by timed events (timing constraints in timed automata)
  - must finish within its (relative) deadline
  - its completion time may influence release of other tasks
Task automata

- Task automata model real time systems with
  - Non-uniformly recurring tasks
  - Single processor

- Timed automata extended with dynamically created tasks

- Task characteristics
  - an abstraction of the computation (best-case and worst-case execution time)
  - generated by timed events (timing constraints in timed automata)
  - must finish within its (relative) deadline
  - its completion time may influence release of other tasks
Task automata

- Task automata model real time systems with
  - Non-uniformly recurring tasks
  - Single processor

- Timed automata extended with dynamically created tasks

- Task characteristics
  - an abstraction of the computation (best-case and worst-case execution time)
  - generated by timed events (timing constraints in timed automata)
  - must finish within its (relative) deadline
  - its completion time may influence release of other tasks
Task automata

- Task automata model real time systems with
  - Non-uniformly recurring tasks
  - Single processor

- Timed automata extended with dynamically created tasks

- Task characteristics
  - an abstraction of the computation (best-case and worst-case execution time)
  - generated by timed events (timing constraints in timed automata)
  - must finish within its (relative) deadline
  - its completion time may influence release of other tasks
Task automata

- Task automata model real time systems with
  - Non-uniformly recurring tasks
  - Single processor

- Timed automata extended with dynamically created tasks

- Task characteristics
  - an abstraction of the computation (best-case and worst-case execution time)
  - generated by timed events (timing constraints in timed automata)
  - must finish within its (relative) deadline
  - its completion time may influence release of other tasks
Task automata

- Task automata model real time systems with
  - Non-uniformly recurring tasks
  - Single processor

- Timed automata extended with dynamically created tasks

- Task characteristics
  - an abstraction of the computation (best-case and worst-case execution time)
  - generated by timed events (timing constraints in timed automata)
  - must finish within its (relative) deadline
  - its completion time may influence release of other tasks
Task automata

- Task automata model real time systems with
  - Non-uniformly recurring tasks
  - Single processor

- Timed automata extended with dynamically created tasks

- Task characteristics
  - an abstraction of the computation (best-case and worst-case execution time)
  - generated by timed events (timing constraints in timed automata)
  - must finish within its (relative) deadline
  - its completion time may influence release of other tasks
Task automata

- Task automata model real time systems with
  - Non-uniformly recurring tasks
  - Single processor

- Timed automata extended with dynamically created tasks

- Task characteristics
  - an abstraction of the computation (best-case and worst-case execution time)
  - generated by timed events (timing constraints in timed automata)
  - must finish within its (relative) deadline
  - its completion time may influence release of other tasks
Schedulability
All tasks are accomplished within their deadlines.

- Schedulability is decidable in certain cases, like
  - non-preemptive scheduling strategy; or
  - best-case = worst-case; or
  - ...
Schedulability
All tasks are accomplished within their deadlines.

- Schedulability is decidable in certain cases, like
  - non-preemptive scheduling strategy; or
  - best-case = worst-case; or
  - ...

Task Automata - Schedulability
Schedulability - Example

A(10) ! 
\( x > 7 \)
\( x := 0 \)

B (6) ! 
\( x > 5 \)
\( x := 0 \)

A :
best-case = 3
worst-case = 4

B :
best-case = 2
worst-case = 5
Schedulability - Example

A(10)!
x > 7
x := 0

B (6)!
x > 5
x := 0

A:
best-case = 3
worst-case = 8

B:
best-case = 2
worst-case = 5
Motivation

Use task automata ideas for schedulability analysis of timed Creol models.
A Simplified Subset

- Reactive asynchronous objects
  - Objects have one processor each

- Communication:
  - Asynchronous message passing
  - Incoming messages are buffered (for each object)

- Computation:
  - No processor release points (methods run to the end)
  - init message in buffer upon creation

- Scheduling strategy (The order of executing incoming messages)
  - Examples: First Come First Served (FCFS), Earliest Deadline First (EDF), Nondeterministic.
  - Preemption is disallowed
A Simplified Subset

- Reactive asynchronous objects
  - Objects have one processor each

- Communication:
  - Asynchronous message passing
  - Incoming messages are buffered (for each object)

- Computation:
  - No processor release points (methods run to the end)
  - init message in buffer upon creation

- Scheduling strategy (The order of executing incoming messages)
  - Examples: First Come First Served (FCFS), Earliest Deadline First (EDF), Nondeterministic.
  - Preemption is disallowed
A Simplified Subset

- Reactive asynchronous objects
  - Objects have one processor each

- Communication:
  - Asynchronous message passing
  - Incoming messages are buffered (for each object)

- Computation:
  - No processor release points (methods run to the end)
  - init message in buffer upon creation

- Scheduling strategy (The order of executing incoming messages)
  - Examples: First Come First Served (FCFS), Earliest Deadline First (EDF), Nondeterministic.
  - Preemption is disallowed
A Simplified Subset

- Reactive asynchronous objects
  - Objects have one processor each

- Communication:
  - Asynchronous message passing
  - Incoming messages are buffered (for each object)

- Computation:
  - No processor release points (methods run to the end)
  - init message in buffer upon creation

- Scheduling strategy (The order of executing incoming messages)
  - Examples: First Come First Served (FCFS), Earliest Deadline First (EDF), Nondeterministic.
  - Preemption is disallowed
A Simplified Subset

- Reactive asynchronous objects
  - Objects have one processor each

- Communication:
  - Asynchronous message passing
  - Incoming messages are buffered (for each object)

- Computation:
  - No processor release points (methods run to the end)
  - init message in buffer upon creation

- Scheduling strategy (The order of executing incoming messages)
  - Examples: First Come First Served (FCFS), Earliest Deadline First (EDF), Nondeterministic.
  - Preemption is disallowed
A Simplified Subset

- Reactive asynchronous objects
  - Objects have one processor each

- Communication:
  - Asynchronous message passing
  - Incoming messages are buffered (for each object)

- Computation:
  - No processor release points (methods run to the end)
  - `init` message in buffer upon creation

- Scheduling strategy (The order of executing incoming messages)
  - Examples: First Come First Served (FCFS), Earliest Deadline First (EDF), Nondeterministic.
  - Preemption is disallowed
A Simplified Subset

- Reactive asynchronous objects
  - Objects have one processor each

Communication:
- Asynchronous message passing
- Incoming messages are buffered (for each object)

Computation:
- No processor release points (methods run to the end)
  - \texttt{init} message in buffer upon creation

Scheduling strategy (The order of executing incoming messages)
- Examples: First Come First Served (FCFS), Earliest Deadline First (EDF), Nondeterministic.
- Preemption is disallowed
A Simplified Subset

- Reactive asynchronous objects
  - Objects have one processor each

- Communication:
  - Asynchronous message passing
  - Incoming messages are buffered (for each object)

- Computation:
  - No processor release points (methods run to the end)
  - \texttt{init} message in buffer upon creation

- Scheduling strategy (The order of executing incoming messages)
  - Examples: First Come First Served (FCFS), Earliest Deadline First (EDF), Nondeterministic.
  - Preemption is disallowed
A Simplified Subset

- Reactive asynchronous objects
  - Objects have one processor each

- Communication:
  - Asynchronous message passing
  - Incoming messages are buffered (for each object)

- Computation:
  - No processor release points (methods run to the end)
  - `init` message in buffer upon creation

- Scheduling strategy (The order of executing incoming messages)
  - Examples: First Come First Served (FCFS), Earliest Deadline First (EDF), Nondeterministic.
  - Preemption is disallowed
A Simplified Subset

- Reactive asynchronous objects
  - Objects have one processor each

Communication:
- Asynchronous message passing
  - Incoming messages are buffered (for each object)

Computation:
- No processor release points (methods run to the end)
  - `init` message in buffer upon creation

Scheduling strategy (The order of executing incoming messages)
- Examples: First Come First Served (FCFS), Earliest Deadline First (EDF), Nondeterministic.
  - Preemption is disallowed
A Simplified Subset

- Reactive asynchronous objects
  - Objects have one processor each

**Communication:**
- Asynchronous message passing
- Incoming messages are buffered (for each object)

**Computation:**
- No processor release points (methods run to the end)
- `init` message in buffer upon creation

**Scheduling strategy (The order of executing incoming messages)**
- Examples: First Come First Served (FCFS), Earliest Deadline First (EDF), Nondeterministic.
- Preemption is disallowed
Example

```plaintext
class C1(worker : C2) begin
  var v1 : boolean;

  op init (w:C2) ==
    worker = w;
    v1 = false;
    ! this.a();
    ! worker.b();

  op a() ==
    if (v1) begin
      worker.b();
    end;
    v1 = ~ v1;
end

class C2 begin
  var v2 : int;

  op init ==
    v2 = 0;

  op b() ==
    v2 = v2 + 1;
    ! caller.a();
end

class main begin
  op init ==
    var r1 : C1;
    var r2 : C2;
    r2 = new C2;
    r1 = new C1(r2);
end
```
Example

```plaintext
class C1(worker : C2) begin
    var v1 : boolean;

    op init(w:C2) ==
    worker = w;
    v1 = false;
    ! this.a();
    ! worker.b();

    op a() ==
    if (v1) begin
        worker.b();
    end;
    v1 = ~ v1;
end

class C2 begin
    var v2 : int;
    op init ==
    v2 = 0;

    op b() ==
    v2 = v2 + 1;
    ! caller.a();
end

class main begin
    op init ==
    var r1 : C1;
    var r2 : C2;
    r2 = new C2;
    r1 = new C1(r2);
end
end
```
Example

class C1(worker : C2) begin
  var v1 : boolean;

  op init(w:C2) ==
    worker = w;
    v1 = false;
    ! this.a();
    ! worker.b();

  op a() ==
    if (v1) begin
      worker.b();
    end;
    v1 = ~ v1;

end

class C2 begin
  var v2 : int;

  op init ==
    v2 = 0;

  op b() ==
    v2 = v2 + 1;
    ! caller.a();
end

class main begin
  op init ==
    var r1 : C1;
    var r2 : C2;
    r2 = new C2;
    r1 = new C1(r2);
end
end
Example

```plaintext
class C1(worker : C2) begin
  var v1 : boolean;

  op init (w:C2) ==
    worker = w;
    v1 = false;
    ! this.a();
    ! worker.b();

  op a() ==
    if (v1) begin
      worker.b();
    end;
    v1 = ~ v1;
end

class C2 begin
  var v2 : int;

  op init ==
    v2 = 0;

  op b() ==
    v2 = v2 + 1;
    ! caller.a();
end

class main begin
  op init ==
    var r1 : C1;
    var r2 : C2;
    r2 = new C2;
    r1 = new C1(r2);
end
end
```
Example

class C1(worker : C2) begin
  var v1 : boolean;
  
  op init(w:C2) ==
    worker = w;
    v1 = false;
    ! this.a();
    ! worker.b();
  end

  op a() ==
    if (v1) begin
      worker.b();
    end;
    v1 = ~ v1;
  end
end

class C2 begin
  var v2 : int;
  
  op init ==
    v2 = 0;
    
    op b() ==
      v2 = v2 + 1;
      ! caller.a();
  end
end

class main begin
  op init ==
    var r1 : C1;
    var r2 : C2;
    r2 = new C2;
    r1 = new C1(r2);
  end
end
class C1( worker : C2) begin
    var v1 : boolean;

    op init(w:C2) ==
        worker = w;
        v1 = false;
        ! this.a();
        ! worker.b();

    op a() ==
        if (v1) begin
            worker.b();
        end;
        v1 = ~ v1;
end

class C2 begin
    var v2 : int;

    op init ==
        v2 = 0;

    op b() ==
        v2 = v2 + 1;
        ! caller.a();
end

class main begin
    op init ==
        var r1 : C1;
        var r2 : C2;
        r2 = new C2;
        r1 = new C1(r2);
end end
```plaintext

class C1(worker : C2) begin
  var v1 : boolean;

  op init(w:C2) ==
    worker = w;
    v1 = false;
    ! this.a();
    ! worker.b();

  op a() ==
    if (v1) begin
      worker.b();
    end;
    v1 = ~ v1;
end

class C2 begin
  var v2 : int;

  op init ==
    v2 = 0;

  op b() ==
    v2 = v2 + 1;
    ! caller.a();
end

class main begin
  op init ==
    var r1 : C1;
    var r2 : C2;
    r2 = new C2;
    r1 = new C1(r2);
end
end
```
Ideas and Problems

Idea:

- Each method can be one task.
- Model task generation patterns with task automata.

But,

- There are more than one processor
- Tasks are specified (as timed automata) rather than execution times
  - and may create new tasks during its execution
- Three categories of method calls (task generation)
  - messages received from other objects
  - messages sent out to other objects
  - self calls
Idea:
- Each method can be one task.
- Model task generation patterns with task automata.

But,
- There are more than one processor
- Tasks are specified (as timed automata) rather than execution times
  - and may create new tasks during its execution
- Three categories of method calls (task generation)
  - messages received from other objects
  - messages sent out to other objects
  - self calls
Ideas and Problems

Idea:

- Each method can be one task.
- Model task generation patterns with task automata.

But,

- There are more than one processor
- Tasks are specified (as timed automata) rather than execution times
  - and may create new tasks during its execution
- Three categories of method calls (task generation)
  - messages received from other objects
  - messages sent out to other objects
  - self calls
Ideas and Problems

Idea:
- Each method can be one task.
- Model task generation patterns with task automata.

But,
- There are more than one processor
- Tasks are specified (as timed automata) rather than execution times
  - and may create new tasks during its execution
- Three categories of method calls (task generation)
  - messages received from other objects
  - messages sent out to other objects
  - self calls
Idea:

- Each method can be one task.
- Model task generation patterns with task automata.

But,

- There are more than one processor

  - Tasks are specified (as timed automata) rather than execution times
    - and may create new tasks during its execution

- Three categories of method calls (task generation)
  - messages received from other objects
  - messages sent out to other objects
  - self calls
Ideas and Problems

Idea:
- Each method can be one task.
- Model task generation patterns with task automata.

But,
- There are more than one processor
- Tasks are specified (as timed automata) rather than execution times
  - and may create new tasks during its execution

- Three categories of method calls (task generation)
  - messages received from other objects
  - messages sent out to other objects
  - self calls
Idea:
- Each method can be one task.
- Model task generation patterns with task automata.

But,
- There are more than one processor
- Tasks are specified (as timed automata) rather than execution times
  - and may create new tasks during its execution
- Three categories of method calls (task generation)
  - messages received from other objects
  - messages sent out to other objects
  - self calls
Example: Object r1

class C1(worker : C2) begin
    var v1 : boolean;

    op init() ==
        v1 = false;
        ! this.a();
        ! worker.b();

    op a() ==
        if (v1) begin
            worker.b();
        end;
        v1 = ~ v1;

end
class C1(worker : C2) begin
    var v1 : boolean;

    op init() ==
        v1 = false;
        ! this.a();
        ! worker.b();

    op a() ==
        if (v1) begin
            worker.b();
        end;
        v1 = ~ v1;
end
Example: Object r1

Driver automaton

\[
\begin{align*}
\text{start} & : x \leq 1 \\
\text{a(4) ?} & : 3 < x_t \\
x_t := 0
\end{align*}
\]
Schedulability of an Object

**Behavior Automaton**

Executing the abstract methods as controlled by the driver automaton

- States: \((s, s'), [B_1, \ldots, B_l]\)
  - \(s\) is the current state of driver automaton
  - \(s'\) is the current state of the currently running method
  - \([B_1, \ldots, B_l]\) is the current queue
  - There is a static bound \(q\) on length of schedulable queues

\[ \Sigma_H = \{!m(d)|m \notin M\} \cup \{?m(d)|m \in M\} \cup \{m|m \in M\} \]

\[ C_H = \bigcup_{i \in [1..n]} C_i \cup C_D \]
Schedulability of an Object

Behavior Automaton
Executing the abstract methods as controlled by the driver automaton

- **States:** \(((s, s'), [B_1, \ldots, B_l])\)
  - \(s\) is the current state of driver automaton
  - \(s'\) is the current state of the currently running method
  - \([B_1, \ldots, B_l]\) is the current queue
  - There is a static bound \(q\) on length of schedulable queues
    - \(q = \frac{\text{max(deadline)}}{\text{min(best-case execution time)}}\)

- \(\Sigma_H = \{!m(d)|m \notin M\} \cup \{?m(d)|m \in M\} \cup \{m|m \in M\}\)
- \(C_H = (\bigcup_{i \in [1..n]} C_i) \cup C_D\)
Schedulability of an Object

Behavior Automaton
Executing the abstract methods as controlled by the driver automaton

- **States:** \( ((s, s'), [B_1, \ldots, B_l]) \)
  - \( s \) is the current state of the **driver** automaton
  - \( s' \) is the current state of the currently running method
  - \([B_1, \ldots, B_l]\) is the current queue
  - There is a static bound \( q \) on length of schedulable queues

- **\( \Sigma_H \):**
  \[ \Sigma_H = \{!m(d) | m \notin M\} \cup \{?m(d) | m \in M\} \cup \{m | m \in M\} \]

- **\( C_H \):**
  \[ C_H = (\bigcup_{i \in [1..n]} C_i) \cup C_D \]
Schedulability of an Object

Behavior Automaton

Executing the abstract methods as controlled by the driver automaton

- States: \(((s, s'), [B_1, \ldots, B_l])\)
  - \(s\) is the current state of the driver automaton
  - \(s'\) is the current state of the currently running method
  - \([B_1, \ldots, B_l]\) is the current queue
  - There is a static bound \(q\) on length of schedulable queues
    - \(q = \frac{\text{max(deadline)}}{\text{min(best-case execution time)}}\)

- \(\Sigma_H = \{!m(d)|m \notin M\} \cup \{?m(d)|m \in M\} \cup \{m|m \in M\}\)
- \(C_H = \bigcup_{i \in [1..n]} C_i \cup C_D\)
Schedulability of an Object

Behavior Automaton

Executing the abstract methods as controlled by the driver automaton

- States: \((s, s'), [B_1, \ldots, B_l]\)
  - \(s\) is the current state of driver automaton
  - \(s'\) is the current state of the currently running method
  - \([B_1, \ldots, B_l]\) is the current queue

- There is a static bound \(q\) on length of schedulable queues
  - \(q = \frac{\text{max(deadline)}}{\text{min(best-case execution time)}}\)

- \(\Sigma_H = \{!m(d)|m \notin M\} \cup \{?m(d)|m \in M\} \cup \{m|m \in M\}\)

- \(C_H = (\bigcup_{i \in [1..n]} C_i) \cup C_D\)
Schedulability of an Object

Behavior Automaton
Executing the abstract methods as controlled by the driver automaton

- States: \(((s, s'), [B_1, \ldots, B_l])\)
  - \(s\) is the current state of the driver automaton
  - \(s'\) is the current state of the currently running method
  - \([B_1, \ldots, B_l]\) is the current queue
  - There is a static bound \(q\) on length of schedulable queues
    - \(q = \max(\text{deadline}) / \min(\text{best-case execution time})\)

\[\Sigma_H = \{!m(d) | m \notin M\} \cup \{?m(d) | m \in M\} \cup \{m | m \in M\}\]

\[C_H = (\bigcup_{i \in [1..n]} C_i) \cup C_D\]
Schedulability of an Object

Behavior Automaton

Executing the abstract methods as controlled by the driver automaton

- **States:** \(((s, s'), [B_1, \ldots, B_l])\)
  - \(s\) is the current state of driver automaton
  - \(s'\) is the current state of the currently running method
  - \([B_1, \ldots, B_l]\) is the current queue
  - There is a static bound \(q\) on length of schedulable queues
    - \(q = \max(\text{deadline}) / \min(\text{best-case execution time})\)

- \(\Sigma_H = \{!m(d) | m \notin M\} \cup \{?m(d) | m \in M\} \cup \{m | m \in M\}\)
- \(C_H = \bigcup_{i \in [1..n]} C_i \cup C_D\)
Schedulability of an Object

Behavior Automaton

Executing the abstract methods as controlled by the driver automaton

- States: \(((s, s'), [B_1, \ldots, B_l])\)
  - \(s\) is the current state of driver automaton
  - \(s'\) is the current state of the currently running method
  - \([B_1, \ldots, B_l]\) is the current queue
  - There is a static bound \(q\) on length of schedulable queues
    - \(q = \frac{\text{max(deadline)}}{\text{min(best-case execution time)}}\)

- \(\Sigma_H = \{!m(d) | m \notin M\} \cup \{?m(d) | m \in M\} \cup \{m | m \in M\}\)

- \(C_H = (\bigcup_{i \in [1..n]} C_i) \cup C_D\)
Schedulability of an Object

Behavior Automaton
Executing the abstract methods as controlled by the driver automaton

- **States:** \(((s, s'), [B_1, \ldots, B_l])\)
  - \(s\) is the current state of driver automaton
  - \(s'\) is the current state of the currently running method
  - \([B_1, \ldots, B_l]\) is the current queue
  - There is a static bound \(q\) on length of schedulable queues
    - \(q = \max(\text{deadline}) / \min(\text{best-case execution time})\)

- \(\Sigma_H = \{!m(d) | m \notin M\} \cup \{?m(d) | m \in M\} \cup \{m | m \in M\}\)

- \(C_H = \bigcup_{i \in [1..n]} C_i \cup C_D\)
Schedulability of an Object

Behavior Automaton

Executing the abstract methods as controlled by the driver automaton

- **States:** \(( (s, s'), [B_1, \ldots, B_l]) \)
  - \(s\) is the current state of driver automaton
  - \(s'\) is the current state of the currently running method
  - \([B_1, \ldots, B_l]\) is the current queue

- There is a static bound \(q\) on length of schedulable queues
  - \(q = \frac{\text{max(deadline)}}{\text{min(best-case execution time)}}\)

- \(\Sigma_H = \{!m(d) | m \notin M\} \cup \{?m(d) | m \in M\} \cup \{m | m \in M\}\)

- \(C_H = \bigcup_{i \in [1..n]} C_i \cup C_D\)
Schedulability of an Object

Behavior Automaton

Executing the abstract methods as controlled by the driver automaton

- **States:** \( ((s, s'), [B_1, \ldots, B_l]) \)
  - \( s \) is the current state of driver automaton
  - \( s' \) is the current state of the currently running method
  - \( [B_1, \ldots, B_l] \) is the current queue
  - There is a static bound \( q \) on length of schedulable queues
    - \( q = \max(\text{deadline}) / \min(\text{best-case execution time}) \)

- **\( \Sigma_H \):** \( \{!m(d) | m \notin M\} \cup \{?m(d) | m \in M\} \cup \{m | m \in M\} \)

- **\( C_H \):** \( \bigcup_{i \in [1..n]} C_i \) \( \cup \) \( C_D \)
Computing the Behavior Automaton

- **Receive**: receiving message indicated in driver automaton
- **Self call**
  - Invocation: when deadline value mentioned explicitly
  - Delegation: inheriting parent task’s deadline
- **External call**
- **Internal**: transitions without sending/receiving messages
- **Context switch**: when current method finishes
- **Queue overflow**: go to error state (keeping model finite)
- **Missed deadline**: from every state to error state
Computing the Behavior Automaton

- **Receive**: receiving message indicated in driver automaton
- **Self call**
  - **Invocation**: when deadline value mentioned explicitly
  - **Delegation**: inheriting parent task’s deadline
- **External call**
- **Internal**: transitions without sending/receiving messages
- **Context switch**: when current method finishes
- **Queue overflow**: go to error state (keeping model finite)
- **Missed deadline**: from every state to error state
Computing the Behavior Automaton

- Receive: receiving message indicated in driver automaton
- Self call
  - Invocation: when deadline value mentioned explicitly
  - Delegation: inheriting parent task’s deadline
- External call
- Internal: transitions without sending/receiving messages
- Context switch: when current method finishes
- Queue overflow: go to error state (keeping model finite)
- Missed deadline: from every state to error state
Computing the Behavior Automaton

- Receive: receiving message indicated in driver automaton
- Self call
  - Invocation: when deadline value mentioned explicitly
  - Delegation: inheriting parent task’s deadline
- External call
- Internal: transitions without sending/receiving messages
- Context switch: when current method finishes
- Queue overflow: go to error state (keeping model finite)
- Missed deadline: from every state to error state
Computing the Behavior Automaton

- Receive: receiving message indicated in driver automaton
- Self call
  - Invocation: when deadline value mentioned explicitly
  - Delegation: inheriting parent task's deadline
- External call
  - Internal: transitions without sending/receiving messages
  - Context switch: when current method finishes
- Queue overflow: go to error state (keeping model finite)
- Missed deadline: from every state to error state
Computing the Behavior Automaton

- Receive: receiving message indicated in driver automaton
- Self call
  - Invocation: when deadline value mentioned explicitly
  - Delegation: inheriting parent task’s deadline
- External call
- Internal: transitions without sending/receiving messages
  - Context switch: when current method finishes
  - Queue overflow: go to error state (keeping model finite)
  - Missed deadline: from every state to error state
Computing the Behavior Automaton

- Receive: receiving message indicated in driver automaton
- Self call
  - Invocation: when deadline value mentioned explicitly
  - Delegation: inheriting parent task’s deadline
- External call
- Internal: transitions without sending/receiving messages
- Context switch: when current method finishes
- Queue overflow: go to error state (keeping model finite)
- Missed deadline: from every state to error state
Computing the Behavior Automaton

- Receive: receiving message indicated in driver automaton
- Self call
  - Invocation: when deadline value mentioned explicitly
  - Delegation: inheriting parent task’s deadline
- External call
- Internal: transitions without sending/receiving messages
- Context switch: when current method finishes
- Queue overflow: go to error state (keeping model finite)
- Missed deadline: from every state to error state
Computing the Behavior Automaton

- **Receive**: receiving message indicated in driver automaton
- **Self call**
  - **Invocation**: when deadline value mentioned explicitly
  - **Delegation**: inheriting parent task’s deadline
- **External call**
- **Internal**: transitions without sending/receiving messages
- **Context switch**: when current method finishes
- **Queue overflow**: go to error state (keeping model finite)
- **Missed deadline**: from every state to error state
Receive

\[(s, s'), (Q, VS) \xrightarrow{m(d)?} ((t, s'), (Q', VS))\]

if \(s \xrightarrow{m(d)?} \overset{c \land L; X, c_i = 0}{D} t\) and

\[(L, c_i, Q') \in \text{sched}(Q, m(d))\) and

\[\text{length}(Q) \leq q\]
Self call: Invocation

\[(s, s'), [B_1, \ldots, B_l], VS) \xrightarrow{m} ((s, t'), Q', VS)\]

if \((s' \xrightarrow{m(d)!} c; X \xrightarrow{B_1} t')\) and \((L, c_i, Q') \in sched([B_1, \ldots, B_l], m(d))\) and \((m \in M)\) and \(l \leq q\)
Self call: Delegation

\[ ((s, s'), [m_1(d, c'), B_2, \ldots, B_l], VS) \xrightarrow{m \in M} ((s, t'), Q', VS) \]

if \((s' \xrightarrow{m_1} t')\) and \((L, Q') \in \text{sched}([m_1(d, c'), \ldots, B_l], m(d, c'))\) and \((m \in M)\) and \(l \leq q\)
External call

\[(s, s'), [B_1, \ldots, B_l], \text{VS} \xrightarrow{m(d)!} \((s, t'), [B_1, \ldots, B_l], \text{VS})\]

if \((s' \xrightarrow{m(d)!} _{c; X} B_1 \ t')\) and

\((m \notin M)\) and

\(l \leq q\)
Internal action

\[(s, s'), [B_1, \ldots, B_l], VS) \xrightarrow{c; X} H ((s, t'), [B_1, \ldots, B_l], VS)\]

if \((s' \xrightarrow{c; X} B_1 t')\) and

\[l \leq q\]
Context-switch

$$((s, s'), [B_1, B_2, \ldots, B_l], VS) \underset{C_l = 0}{\overset{H}{\rightarrow}} ((s, start(B_2)), [B_2, \ldots, B_l], VS)$$

if $$s' \in \text{final}(B_1)$$ and

$$C_l = \text{local\_clocks}(B_2)$$ and

$$l \leq q$$

The source state is “urgent”.
Overflow

\[(s, s'), [B_1, \ldots, B_l], VS) \rightarrow_h \text{ error if } (l > q)\]
Check deadline for every message in the queue.

\[( (s, s'), [m_i(d_i, c_i), \ldots], VS) \xrightarrow{(c_i < d_i)}_{error} H \]
How we do it

Mahdi Jaghoori (CWI)
Compatibility Checking

- After each object is checked to be schedulable
  - with respect to its driver
- Is a complete system containing these objects schedulable?
  - Does the environment of each object respect the driver?
- The composition of drivers should be an over-approximation of the real system
- What does over-approximation mean?
  - A message $m$ is sent to $r1$ only if expected by its driver (within the correct time interval); and,
  - $!m(d')$ matches $?m(d)$ in the driver of $r1$ only if $d \leq d'$
    - $?m(d)$ in driver: $m$ can be finished within $d$ time units
    - $!m(d')$ in the system: requiring that $m$ should finish within $d'$ time units
Compatibility Checking

- After each object is checked to be schedulable with respect to its driver

- Is a complete system containing these objects schedulable?
  - Does the environment of each object respect the driver?

- The composition of drivers should be an over-approximation of the real system

- What does over-approximation mean?
  - A message \( m \) is sent to \( r1 \) only if expected by its driver (within the correct time interval); and,
  - \( !m(d') \) matches \(?m(d)\) in the driver of \( r1 \) only if \( d \leq d' \)
    - \( ?m(d) \) in driver: \( m \) can be finished within \( d \) time units
    - \( !m(d') \) in the system: requiring that \( m \) should finish within \( d' \) time units
Compatibility Checking

- After each object is checked to be schedulable with respect to its driver
- Is a complete system containing these objects schedulable?
  - Does the environment of each object respect the driver?
- The composition of drivers should be an over-approximation of the real system
- What does over-approximation mean?
  - A message $m$ is sent to $r1$ only if expected by its driver (within the correct time interval); and,
  - $!m(d')$ matches $?m(d)$ in the driver of $r1$ only if $d \leq d'$
    - $!m(d)$ in driver: $m$ can be finished within $d$ time units
    - $?m(d')$ in the system: requiring that $m$ should finish within $d'$ time units
Compatibility Checking

- After each object is checked to be schedulable with respect to its driver
- Is a complete system containing these objects schedulable?
  - Does the environment of each object respect the driver?

- The composition of drivers should be an over-approximation of the real system

- What does over-approximation mean?
  - A message $m$ is sent to $r_1$ only if expected by its driver (within the correct time interval); and,
  - $! m(d')$ matches $? m(d)$ in the driver of $r_1$ only if $d \leq d'$
    - $! m(d)$ in driver: $m$ can be finished within $d$ time units
    - $! m(d')$ in the system: requiring that $m$ should finish within $d'$ time units
Compatibility Checking

- After each object is checked to be schedulable
  - with respect to its driver
- Is a complete system containing these objects schedulable?
  - Does the environment of each object respect the driver?

- The composition of drivers should be an over-approximation of the real system

- What does over-approximation mean?
  - A message $m$ is sent to $r1$ only if expected by its driver (within the correct time interval); and,
  - $!m(d')$ matches $?m(d)$ in the driver of $r1$ only if $d \leq d'$
    - $!m(d)$ in driver: $m$ can be finished within $d$ time units
    - $!m(d')$ in the system: requiring that $m$ should finish within $d'$ time units
Compatibility Checking

- After each object is checked to be schedulable
  - with respect to its driver
- Is a complete system containing these objects schedulable?
  - Does the environment of each object respect the driver?

- The composition of drivers should be an over-approximation of the real system

- What does over-approximation mean?
  - A message $m$ is sent to $r1$ only if expected by its driver (within the correct time interval); and,
  - $!m(d')$ matches $?m(d)$ in the driver of $r1$ only if $d \leq d'$
    - $?m(d)$ in driver: $m$ can be finished within $d$ time units
    - $!m(d')$ in the system: requiring that $m$ should finish within $d'$ time units
Compatibility Checking

- After each object is checked to be schedulable with respect to its driver
- Is a complete system containing these objects schedulable?
  - Does the environment of each object respect the driver?

- The composition of drivers should be an over-approximation of the real system

- What does over-approximation mean?
  - A message $m$ is sent to $r1$ only if expected by its driver (within the correct time interval); and,
  - $!m(d')$ matches $?m(d)$ in the driver of $r1$ only if $d \leq d'$
    - $?m(d)$ in driver: $m$ can be finished within $d$ time units
    - $!m(d')$ in the system: requiring that $m$ should finish within $d'$ time units
Compatibility Checking

- After each object is checked to be schedulable with respect to its driver
- Is a complete system containing these objects schedulable?
  - Does the environment of each object respect the driver?

- The composition of drivers should be an over-approximation of the real system

- What does over-approximation mean?
  - A message $m$ is sent to $r1$ only if expected by its driver (within the correct time interval); and,
  - $!m(d')$ matches $?m(d)$ in the driver of $r1$ only if $d \leq d'$
    - $?m(d)$ in driver: $m$ can be finished within $d$ time units
    - $!m(d')$ in the system: requiring that $m$ should finish within $d'$ time units
Compatibility Checking

- After each object is checked to be schedulable with respect to its driver
- Is a complete system containing these objects schedulable?
  - Does the environment of each object respect the driver?
- The composition of drivers should be an over-approximation of the real system

What does over-approximation mean?
- A message $m$ is sent to $r1$ only if expected by its driver (within the correct time interval); and,
- $!m(d')$ matches $?m(d)$ in the driver of $r1$ only if $d \leq d'$
  - $?m(d)$ in driver: $m$ can be finished within $d$ time units
  - $!m(d')$ in the system: requiring that $m$ should finish within $d'$ time units
Compatibility Checking

- After each object is checked to be schedulable with respect to its **driver**
- Is a complete system containing these objects schedulable?
  - Does the environment of each object respect the driver?
- The composition of drivers should be an over-approximation of the real system
- **What does over-approximation mean?**
  - A message $m$ is sent to $r1$ only if expected by its driver (within the correct time interval); and,
  - $!m(d')$ matches $?m(d)$ in the driver of $r1$ only if $d \leq d'$
    - $?m(d)$ in driver: $m$ can be finished within $d$ time units
    - $!m(d')$ in the system: requiring that $m$ should finish within $d'$ time units
Test Compatibility

- The product of all drivers \((A)\) should be an abstraction (over-approximation) of the whole system \((S)\)
- Take a (timed) trace in \(A\)
  - Using UPPAAL model checker
  - Check the negation of a desired property
- Add to each state a transition
  - to aFAIL state
  - catching incompatible behaviors
- If FAIL is not reachable, it is inconclusive.
The product of all drivers ($A$) should be an abstraction (over-approximation) of the whole system ($S$)

- Take a (timed) trace in $A$
  - Using UPPAAL model checker
  - Check the negation of a desired property
- Add to each state a transition
  - to a FAIL state
  - catching incompatible behaviors
- If FAIL is not reachable, it is inconclusive.
Test Compatibility

- The product of all drivers ($A$) should be an abstraction (over-approximation) of the whole system ($S$)
- Take a (timed) trace in $A$
  - Using UPPAAL model checker
  - Check the negation of a desired property
- Add to each state a transition
  - to a FAIL state
  - catching incompatible behaviors
- If FAIL is not reachable, it is inconclusive.
Test Compatibility

- The product of all drivers \((A)\) should be an abstraction (over-approximation) of the whole system \((S)\)
- Take a (timed) trace in \(A\)
  - Using UPPAAL model checker
  - Check the negation of a desired property
- Add to each state a transition
  - to a FAIL state
  - catching incompatible behaviors
- If FAIL is not reachable, it is inconclusive.
Test Compatibility

- The product of all drivers ($A$) should be an abstraction (over-approximation) of the whole system ($S$)
- Take a (timed) trace in $A$
  - Using UPPAAL model checker
  - Check the negation of a desired property
- Add to each state a transition
  - to a FAIL state
  - catching incompatible behaviors
- If FAIL is not reachable, it is inconclusive.
Test Compatibility

- The product of all drivers \((A)\) should be an abstraction (over-approximation) of the whole system \((S)\)
- Take a (timed) trace in \(A\)
  - Using UPPAAL model checker
  - Check the negation of a desired property
- Add to each state a transition
  - to a FAIL state
    - catching incompatible behaviors
- If FAIL is not reachable, it is inconclusive.
The product of all drivers \((A)\) should be an abstraction (over-approximation) of the whole system \((S)\).

Take a (timed) trace in \(A\)
- Using UPPAAL model checker
- Check the negation of a desired property

Add to each state a transition
- to a FAIL state
- catching incompatible behaviors

If FAIL is not reachable, it is inconclusive.
Test Compatibility

- The product of all drivers \((A)\) should be an abstraction (over-approximation) of the whole system \((S)\)
- Take a (timed) trace in \(A\)
  - Using UPPAAL model checker
  - Check the negation of a desired property
- Add to each state a transition
  - to a FAIL state
  - catching incompatible behaviors
- If FAIL is not reachable, it is inconclusive.
Conclusions and Future Work

- We adjusted task automata for a subset of Creol
  - Considering self-calls/delegation
  - Tasks specified (instead of best and worst-case execution times)
- Schedulability analyzed for each class
  - The expected use pattern modeled in driver
- For a complete system, compatibility is tested.

Future work
- Schedulability analysis for complete Creol language
  - Synchronous communication
  - Processor release points
- Scheduler specification