

Lecture 13: Outline

- Static verification (part 2)
- Model checking of temporal and timing properties
- The Uppaal model checker

Some Spec# constructs

- \Rightarrow short-circuiting implication
- \Leftrightarrow if and only if
- **result** denotes method return value
- **old**(E) denotes E evaluated in method's pre-state
- **requires** E; declares precondition
- **ensures** E; declares postcondition
- **modifies** w; declares what a method is allowed to modify
- **assert** E; in-line assertion

Examples:

- **forall** {**int** k **in** (0: a.Length); a[k] > 0};
- **exists** {**int** k **in** (0: a.Length); a[k] > 0};
- **exists unique** {**int** k **in** (0: a.Length); a[k] > 0};

```
void Square(int[]! a)  
  modifies a[*];  
  ensures forall{int i in (0: a.Length); a[i] == i*i};
```

Examples:

- **sum** {**int** k **in** (0: a.Length); a[k]};
- **product** {**int** k **in** (1..n); k};
- **min** {**int** k **in** (0: a.Length); a[k]};
- **max** {**int** k **in** (0: a.Length); a[k]};
- **count** {**int** k **in** (0: n); a[k] % 2 == 0};

Intervals:

- The half-open interval {**int** i **in** (0: n)}
- means i satisfies $0 \leq i < n$
- The closed (inclusive) interval {**int** k **in** (0..n)}
- means i satisfies $0 \leq i \leq n$

Comprehensions in Spec# (cont.)

We may also use **filters**:

- `sum {int k in (0: a.Length), 5<=k; a[k]};`
- `product {int k in (0..100), k % 2 == 0; k};`

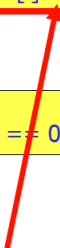
Note that the following two expressions are equivalent:

- `sum {int k in (0: a.Length), 5<=k; a[k]};`
- `sum {int k in (5: a.Length); a[k]};`

Loop Invariants (cont.)

```
public static int SumEvens(int[] a)
ensures result == sum{int i in (0: a.Length) a[i] % 2 == 0; a[i]};
{
  int s = 0;
  for (int n = 0; n < a.Length; n++)
    invariant n <= a.Length;
    invariant s == sum{int i in (0:n), a[i] % 2 == 0; a[i]};

    {
      if (a[n] % 2 == 0)
      {
        s += a[n];
      }
    }
  return s;
}
```



Filters the even values
From the quantified range

Invariant variations: Sum0

```
public static int Sum0(int[] a)
ensures result == sum{int i in (0 : a.Length); a[i]};
{  int s = 0;
   for (int n = 0; n < a.Length; n++)
     invariant n <= a.Length && s == sum{int i in (0: n); a[i]};
   {
       s += a[n];
   }
   return s;
}
```

This loop invariant
focuses on what has
been summed so far.

Invariant variations: Sum1

```
public static int Sum1(int[] a)
ensures result == sum{int i in (0 : a.Length); a[i] };
{
  int s = 0;
  for (int n = 0; n < a.Length; n++)
    invariant n <= a.Length &&
      s + sum{int i in (n: a.Length); a[i]}
        == sum{int i in (0: a.Length); a[i]}
    {
      s += a[n];
    }
  return s;
}
```

This loop invariant focuses on what is yet to be summed.

The *count* Quantifier

```
public int Counting(int[] a)
    ensures result == count{int i in (0: a.Length); a[i] == 0};
{
    int s = 0;
    for (int n = 0; n < a.Length; n++)
        invariant n <= a.Length;
        invariant s == count{int i in (0: n); a[i] == 0};
    {
        if (a[n]== 0) s = s + 1;
    }
    return s;
}
```

Counts the number of
0's in an int [] a;

The *min* Quantifier

```
public int Minimum()
```

```
    ensures result == min{int i in (0: a.Length); a[i]};
```

```
{
```

```
    int m = System.Int32.MaxValue;
```

```
    for (int n = 0; n < a.Length; n++)
```

```
        invariant n <= a.Length;
```

```
        invariant m == min{int i in (0: n); a[i]};
```

```
    {
```

```
        if (a[n] < m)
```

```
            m = a[n];
```

```
    }
```

```
}
```

```
return m;
```

```
}
```

Calculates the minimum value
in an int []! a;

The *max* Quantifier

```
public int MaxEven()  
ensures result == max{int i in (0: a.Length), a[i] % 2 == 0; a[i]};  
{  
    int m = System.Int32.MinValue;  
    for (int n = 0; n < a.Length; n++)  
        invariant n <= a.Length;  
        invariant m == max{int i in (0: n), a[i] % 2 == 0; a[i]};  
    {  
        if (a[n] % 2 == 0 && a[n] > m)  
            m = a[n];  
    }  
    return m;  
}
```

Calculates the maximum even value in an int []! a;

How to help the verifier ...

Recommendations when using comprehensions:

- Write specifications in a form that is as close to the code as possible.
- When writing loop invariants, write them in a form that is as close as possible to the postcondition

In our *SegSum* example where we summed the array elements `a[i] ... a[j-1]`, we could have written the postcondition in either of two forms:

```
ensures result == sum{int k in (i: j); a[k]};
```

```
ensures result ==
```

```
    sum{int k in (0: a.Length), i <= k && k < j; a[k]};
```

How to help the verifier ... (cont.)

Recommendation: When writing loop invariants, write them in a form that is as close as possible to the postcondition.

```
ensures result == sum{int k in (i: j); a[k]};
```

```
invariant i <= n && n <= j;
```

```
invariant s == sum{int k in (i: n); a[k]};
```

OR

```
ensures result ==
```

```
    sum{int k in (0: a.Length), i <= k && k < j; a[k]};
```

```
invariant 0 <= n && n <= a.Length;
```

```
invariant s == sum{int k in (0: n), i <= k && k < j; a[k]};
```

- Specifying the rules for using methods is achieved through contracts, which spell out what is expected of the caller (**preconditions**) and what the caller can expect in return from the implementation (**postconditions**).
- To specify the design of an implementation, we use an assertion involving the data in the class called an **object invariant**.
- Each objects data fields must satisfy the invariant at all **stable** times

Object invariant example

```
public class RockBand
{
    int shows;
    int ads;

    invariant shows <= ads;
    public void Play()
    {
        ads++;
        shows++;
    }
}
```

<RockBand1.ssc>

Broken object invariant example

```
public class RockBand
{
    int shows;
    int ads;

    invariant shows <= ads;
    public void Play()
    {
        shows++;
        ads++;
    }
}
```


Broken object invariant example (cont.)

```
public class RockBand
```

RockBand2.ssc(13,5): Error: Assignment to field
RockBand.shows of non-exposed target object may
break invariant: shows <= ads

Spec# program verifier finished with 4 verified, 1 error

```
    shows++;  
    ads++;  
  }  
}
```

Fixed object invariant example

```
public class RockBand
{
    int shows;
    int ads;
    invariant shows <= ads;
    public void Play()
    {
        expose(this)
        {
            shows++;
            ads++;
        }
    }
}
```

Method Reentrancy

```
public class RockBand
{
    int shows;
    int ads;
    invariant shows <= ads;
    public void Play()
    {
        expose(this)
        {
            shows++;
            Play();
            ads++;
        }
    }
}
```

Method Reentrancy (cont.)

Verifying RockBand.Play ...
RockBand4.ssc(20,3): Error:

The call to `RockBand.Play()` requires target object to be peer consistent

```

    {
        {
            shows++;
            Play();
            ads++;
        }
    }
}

```

Method Reentrancy (cont.)

```
public class RockBand
{
    int shows;
    int ads;
    invariant shows <= ads;
    public void Play()
    {
        expose(this)
        {
            shows++;
            ads++;
        }
        Play();
    }
}
```

<RockBand6.ssc>

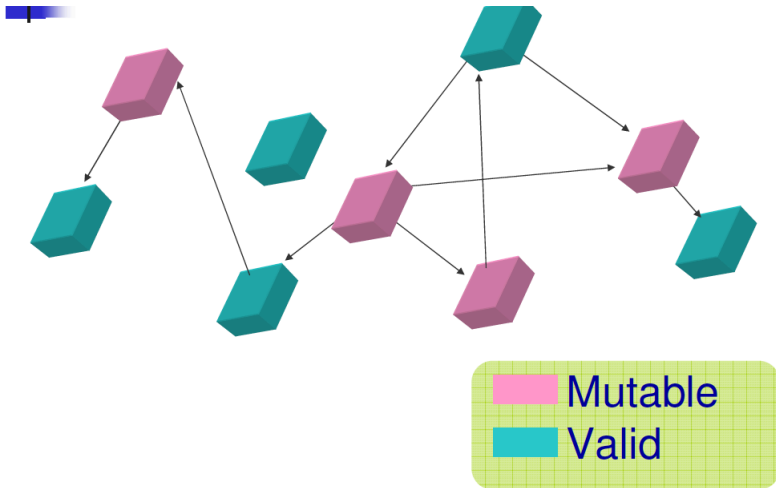
■ Mutable

- Object invariant might be violated
- Field updates are allowed

■ Valid

- Object invariant holds
- Field updates allowed only if they maintain the invariant

Object States (cont.)



To Mutable and Back

```
public class RockBand
```

```
{  int shows;
```

```
    int ads;
```

```
    invariant shows <= ads;
```

```
    public void Play()
```

```
        modifies shows, ads;
```

```
        ensures ads== old(ads)+1 && shows ==old(shows)+1
```

```
    {  expose(this) {
```

```
        shows++;  
        ads ++;
```

```
    }
```

```
}
```

```
}
```

changes **this**
from valid to mutable

can update ads and shows
because **this.mutable**

changes **this**
from mutable to valid

To Mutable and Back

```
class Counter{
```

```
  int c;
```

```
  bool even;
```

```
  invariant 0 <= c;
```

```
  invariant even <==> c % 2 == 0;
```

```
  ...
```

```
  public void Inc ()
```

```
    modifies c;
```

```
    ensures c == old(c)+1:
```

```
  {   expose(this) {
```

```
    c ++;  
    even = !even ;
```

```
  }
```

```
}
```

```
}
```

changes **this**
from valid to mutable

can update c and even,
because **this.mutable**

changes **this**
from mutable to valid

Object Invariants: Summary

```
class Counter{
    int c;
    bool even;
    invariant 0 <= c;
    invariant even <==> c % 2 == 0;

    public Counter()
    {
        c= 0;
        even = true;
    }

    public void Inc ()
    modifies c;
    ensures c == old(c)+1;
    {
        expose (this) {
            c++;
            even = !even ;
        }
    }
}
```

The invariant may be broken in the constructor

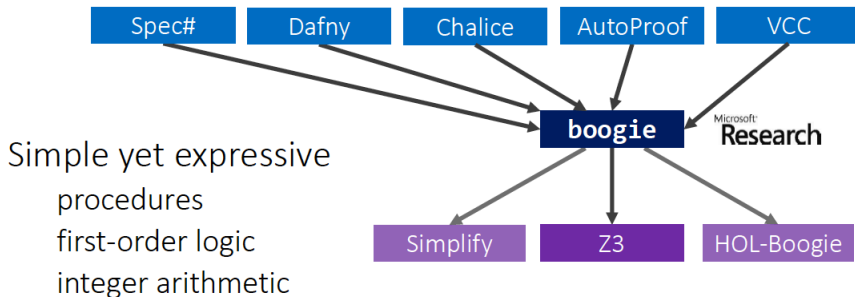
The invariant must be established & checked after construction

The object invariant may be broken within an expose block

Specification inheritance

- Spec# verifies a call to a virtual method M against the specification of M in the static type of the receiver and enforces that all overrides of M in subclasses live up to that specification
- An overriding method inherits the precondition, postcondition, and **modifies** clause from the methods it overrides.
- It may declare additional postconditions, but not additional preconditions or **modifies** clauses because a stronger precondition or a more permissive **modifies** clause would come as a surprise to a caller of the superclass method

The Boogie Intermediate Verifier/Verification Language



Some Research Languages that use Boogie as an Intermediate Language

- **Chalice**: specification and verification of concurrent programs using shared memory and mutual exclusion via locks. Chalice supports dynamic object creation, dynamic thread creation (fork and join), mutual-exclusion and readers-writers locks, monitor invariants, thread pre- and postconditions
- **Dafny**: an object-based language where specifications are written in the style of dynamic frames. The Dafny programming language is designed to support the static verification of programs. It is imperative, sequential, supports generic classes, dynamic allocation, and inductive datatypes, and builds in specification constructs

Some Research Languages that use Boogie as an Intermediate Language

- **AutoProof**, a verification tool that translates Eiffel programs to Boogie and uses the Boogie verifier to prove them. AutoProof fully supports several advanced object-oriented features including polymorphism, inheritance, and function objects
- **VCC** is a tool that proves correctness of annotated concurrent C programs or finds problems in them. VCC extends C with design by contract features, like pre- and postcondition as well as type invariants
- **HAVOC** is a tool for specifying and checking properties of systems software written in C, in the presence of pointer manipulations, unsafe casts and dynamic memory allocation. The assertion logic of HAVOC allows the expression of properties of linked lists and arrays

- Strongly influenced by Spec#;
Focus on runtime verification and testing
- **Code Contracts** provide a language-agnostic way to express coding assumptions in .NET programs. The contracts take the form of preconditions, postconditions, and object invariants
- Contracts act as checked documentation of your external and internal APIs. The contracts are used to improve testing via runtime checking, enable static contract verification, and documentation generation
- Improved testability: each contract acts as an oracle, giving a test run a pass/fail indication; automatic testing tools can take advantage of contracts to generate more meaningful unit tests

Verification by model checking (reminder)

- Model checking usually relies on a generated *reachability graph* of system states
- A model checker explores all the state traces in the graph and checks the property (e.g. invariant preservation) in reached states
- Since a model checker operates on traces (not just single states), additional reachability properties can be formulated and checked
- For instance, that eventually some specific state will be reached or occurrence of some particular states will eventually lead to specific required states

$critical_fault = TRUE \longrightarrow alarm = ON \wedge shutdown_mode = TRUE$

Model checking of temporal and timing system properties

- One of the advantages of model checking is a possibility to check eventual reachability of certain desirable states
- Such properties are usually called liveness (in contrast, invariant properties are often called safety ones)
- Often, we need to enforce a time constraint on such eventual reachability
- Example, "if critical fault occurs, then, within the required time limit T , a desired safe state should be reached by the system"
- To reason about such properties, we need to introduce real time into our system models

The Uppaal tool

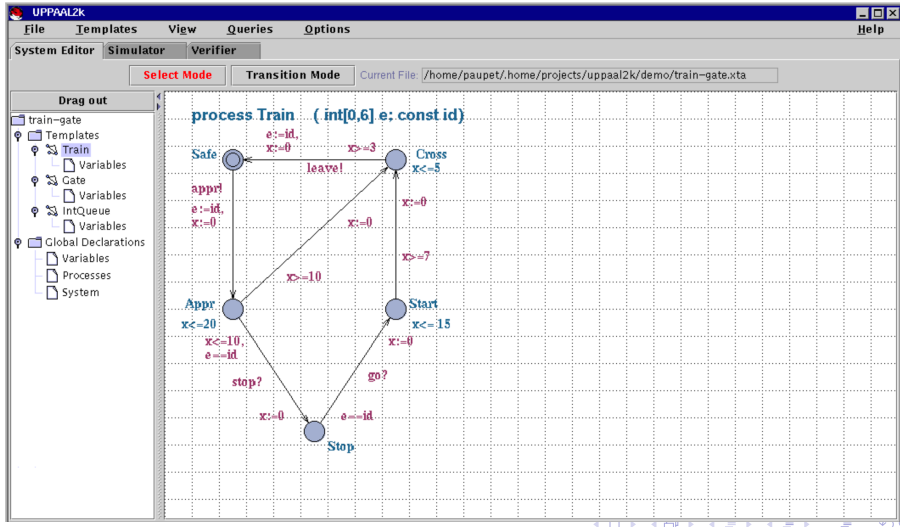
- Uppaal – integrated tool environment for modelling, simulation and verification of real-time systems
- Developed jointly by Uppsala (Sweden) and Aalborg (Denmark) universities
- It is appropriate for the systems that can be modelled as a collection of non-deterministic processes (timed automata) with finite control structure and real-valued clocks, communicating through channels or shared variables
- Typical application areas: real-time controllers, communication protocols, etc.

The Uppaal tool (cont.)

- Uppaal consists of 3 main parts: a description language, a simulator, and a model-checker
- The Uppaal language allows us to describe (model) the system behaviour as networks of automata extended with clocks and data variables. Creating of such models is facilitated by a graphical system editor
- The simulator is a validation tool which enables examination of possible dynamic system executions
- The model checker does exhaustive checking of the dynamic system behaviour and allows to verify invariant and liveness (reachability) properties of a system

The Uppaal graphical editor

Allows modelling a system process as an automata. The circles and arrows represent system states and transitions. Time progress – within in a state



The Uppaal simulator

Provides visualisation of possible dynamic behaviours. Also shows traces (counter-examples) generated by the model checker

The screenshot displays the UPPAAL2K simulator interface. The top menu bar includes File, Templates, View, Queries, Options, and Help. The main window is divided into several panels:

- System Editor:** Contains tabs for System Editor, Simulator, and Verifier. The Enabled Transitions list shows: (Train2.trans4, Gate.trans3), (Train3.trans4, Gate.trans3), and (Train4.trans3, Gate.trans4). Below this are Next and Reset buttons.
- Simulation Trace:** Displays a sequence of events: (Safe, Safe, Safe, Safe, s2, Start), (Train4.trans4, Gate.trans8), (Safe, Safe, Safe, Appr, s3, Start), (Gate.trans9, Queue.trans5), (Safe, Safe, Safe, Appr, Occ1, Start), (Train4.trans1), (Safe, Safe, Safe, Cross, Occ1, Start), (Train1.trans4, Gate.trans3), (Appr, Safe, Safe, Cross, Occ2, Start), (Train1.trans2, Gate.trans5), (Stop, Safe, Safe, Cross, s4, Start), and (Gate.trans10, Queue.trans5).
- Variables:** Lists variables and their values: $e1 = 1$, $Queue.len = 4$, $Queue.list[0] = 1$, $Queue.list[1] = 0$, $Queue.list[2] = 0$, $Queue.list[3] = 0$, $Queue.list[4] = 0$, $Queue.list[5] = 2$, $Queue.i = 0$, $Train1.x \text{ in } [0,5]$, $Train2.x \text{ in } [10,inf]$, $Train3.x \text{ in } [10,inf]$, $Train4.x \text{ in } [0,5]$, $Train2.x - Train1.x$, $Train3.x - Train1.x$, $Train4.x - Train1.x$, $Train3.x - Train2.x$, $Train4.x - Train2.x$, and $Train4.x - Train3.x$.
- State Transition Diagrams:** Four diagrams are shown, each representing a process:
 - process Train1:** Shows states Safe, Appr, Cross, and Stop. Transitions are labeled with events like $e1 := 1$, $x := 0$, $x := 3$, $x := 5$, $x := 0$, $x := 7$, $x := 10$, $x := 15$, $x := 0$, $e1 := 1$, and $x := 0$.
 - process Train2:** Similar to Train1, with states Safe, Appr, Cross, and Stop. Transitions are labeled with events like $e1 := 2$, $x := 0$, $x := 3$, $x := 5$, $x := 0$, $x := 7$, $x := 10$, $x := 15$, $x := 0$, $e1 := 2$, and $x := 0$.
 - process Gate:** Shows states Safe, Appr, Cross, and Stop. Transitions are labeled with events like $e1 := 1$, $x := 0$, $x := 3$, $x := 5$, $x := 0$, $x := 7$, $x := 10$, $x := 15$, $x := 0$, $e1 := 1$, and $x := 0$.
 - process Queue:** Shows states Safe, Appr, Cross, and Stop. Transitions are labeled with events like $e1 := 1$, $x := 0$, $x := 3$, $x := 5$, $x := 0$, $x := 7$, $x := 10$, $x := 15$, $x := 0$, $e1 := 1$, and $x := 0$.

At the bottom, there is a Trace File input field and buttons for Prev, Next, Replay, Open, Save, and Random. A speed slider is also present, ranging from Slow to Fast.

The Uppaal model checker

Allows to formulate and verify invariant/safety and reachability/liveness system properties by exploring the system state space

The screenshot shows the UPPAAL2K model checker interface. The main window is titled "UPPAAL2K" and has a menu bar with "File", "Templates", "View", "Queries", "Options", and "Help". Below the menu bar are three tabs: "System Editor", "Simulator", and "Verifier". The "System Editor" tab is active, showing a list of processes (P0 to P7) with their respective code snippets. To the right of the process list are three buttons: "Model Check", "Insert", and "Remove". Below the process list is a "Query" section with a text area containing "E<> Gate.Occ1". Below the query section is a "Comment" section with a text area containing "Gate can receive (and store in queue) msg's from approaching trains." At the bottom of the window is a "Status" section showing the results of the model check. The status section contains the following text: "Property is satisfied.", "E<> (Train1.Cross and Train2.Stop and Train3.Stop and Train4.Stop)", "Property is satisfied.", "A[] not((Train1.Cross and (Train2.Cross or Train3.Cross or Train4.Cross)) \ or (Train2.Cross and (Train3.Cross or Train4.Cross)) \ or (Train3.Cross and (Train4.Cross or Train3.Cross or Train2.Cross)))", and "Property is satisfied." The status section also has a scrollbar on the right.

UPPAAL2K

File Templates View Queries Options Help

System Editor Simulator Verifier

Overview

P0

→ P1 E<> Gate.Occ1

→ P2 E<> Train1.Cross

P3 E<> Train2.Cross

→ P4 E<> (Train1.Cross and Train2.Stop)

P5 E<> (Train1.Cross and Train2.Stop and Train3.Stop and Train4.Stop)

P6

→ P7 A[] not((Train1.Cross and (Train2.Cross or Train3.Cross or Train4.Cross)) \ or (Train2.Cross and (

Model Check

Insert

Remove

Comments

Query

E<> Gate.Occ1

Comment

Gate can receive (and store in queue) msg's from approaching trains.

Status

Property is satisfied.

E<> (Train1.Cross and Train2.Stop and Train3.Stop and Train4.Stop)

Property is satisfied.

A[] not((Train1.Cross and (Train2.Cross or Train3.Cross or Train4.Cross)) \ or (Train2.Cross and (Train3.Cross or Train4.Cross)) \ or (Train3.Cross and (Train4.Cross or Train3.Cross or Train2.Cross))

Property is satisfied.

Patterns of checked temporal properties

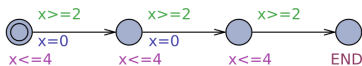
- $E<> p$: there exists an execution path where p eventually holds;
- $A[] p$: for all paths p always holds (invariant);
- $E[] p$: there is an execution path where p always holds;
- $A<> p$: for all paths p will eventually hold;
- $p \rightarrow q$: whenever p holds, q will eventually hold.

The Uppaal SMC tool

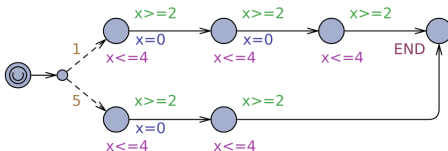
- Recently, the Uppaal tool was extended to support statistical model checking (SMC), where properties are checked with some statistical confidence (probabilities)
- The Uppaal language for modelling real-time systems has also been extended with probabilistic transition branching and clock rates (probabilistic distribution)
- The exhaustive model checking of the system state space is replaced with simulating the system behaviour for some number of "system runs" and accumulating the statistical results
- Timed automata are thus augmented to become stochastic timed automata

The Uppaal SMC tool (cont.)

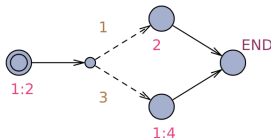
The language is extended with probabilistic transition branching (in 2nd and 3rd diagrams) and clock rates (in 3rd diagram)



(a) A_1 .

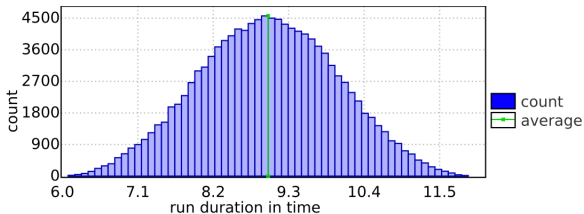


(b) A_2 .

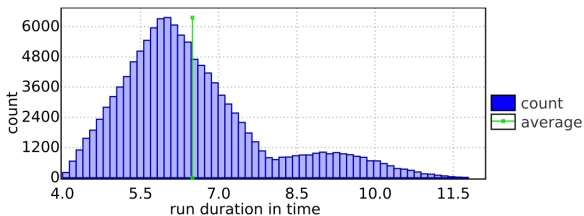


The Uppaal SMC tool (cont.)

Visualisation of distribution of reachability time



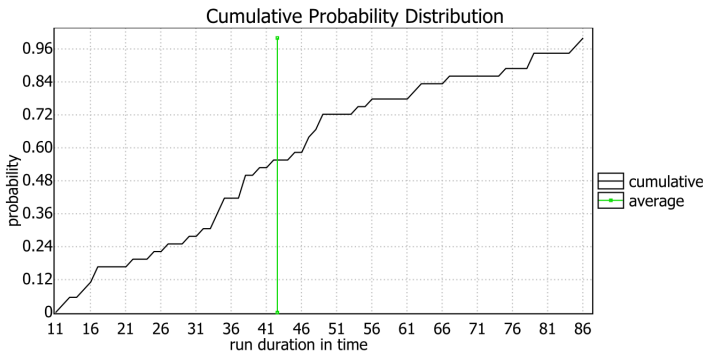
(a) A_1 arrival to **END**.



(b) A_2 arrival to **END**.

The Uppaal SMC tool (cont.)

The cumulative probability distribution of reaching a desired state within the given time



Parameters: $\alpha=0.05$, $\epsilon=0.05$, bucket width=1, bucket count=75

Runs: 36 in total, 36 (100%) displayed, 0 (0%) remaining

Span of displayed sample: [11.0447, 85.2899]

Mean of displayed sample: 42.5735 ± 7.10607 (95% CI)