eXtended Reactive Modules

Benoît Sigoure
<benoit.sigoure@lrde.epita.fr>

EPITA Research and Development Laboratory

June 14, 2006

Outline

1. Motivation
   - Introduction: PRISM and Reactive Modules
   - Typical example: A sensor network
   - eXtended Reactive Modules' solution
2. eXtended Reactive Modules' features
   - The package
   - xrm-front's features
3. Summary

Outline

Model-checking, (Reactive) Modules and PRISM

- Reactive Modules is a formalism.
- PRISM is a probabilistic model checker.
- APMC is an Approximate Probabilistic Model Checker.

Model-checking, (Reactive) Modules and PRISM

- Reactive Modules is a formalism.
  - Used to describe concurrent systems.
- PRISM is a probabilistic model checker.
- APMC is an Approximate Probabilistic Model Checker.

Model-checking, (Reactive) Modules and PRISM

- Reactive Modules is a formalism.
  - Used to describe concurrent systems.
  - Ideal for model-checking.
- PRISM is a probabilistic model checker.

- APMC is an Approximate Probabilistic Model Checker.

Model-checking, (Reactive) Modules and PRISM

- Reactive Modules is a formalism.
  - Used to describe concurrent systems.
  - Ideal for model-checking.
- PRISM is a probabilistic model checker.
  - Introduces the PRISM language...
  - ... which is based on Reactive Modules' syntax.
  - Widely used.
- APMC is an Approximate Probabilistic Model Checker.

Model-checking, (Reactive) Modules and PRISM

- Reactive Modules is a formalism.
  - Used to describe concurrent systems.
  - Ideal for model-checking.
- PRISM is a probabilistic model checker.
  - Introduces the PRISM language...
  - ... which is based on Reactive Modules' syntax.
  - Widely used.
- APMC is an Approximate Probabilistic Model Checker.
  - Uses PRISM's parser.
  - Can handle very large systems.

The PRISM language

Main problem: describing large modules is almost impossible using the PRISM language.

Module renaming

```
module process1
  x1 : [0..1];
  [] (x1=x5) -> 0.5 : (x1'=0) + 0.5 : (x1'=1);
  [] !x1=x5 -> (x1'=x5);
endmodule

// Add further processes through renaming.
module process2 = process1[x1=x2, x5=x1] endmodule
module process3 = process1[x1=x3, x5=x2] endmodule
module process4 = process1[x1=x4, x5=x3] endmodule
module process5 = process1[x1=x5, x5=x4] endmodule
```

Several limitations

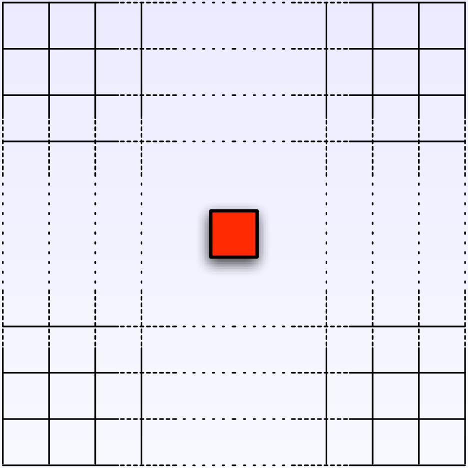
- Imagine the previous example with 100 (or more) modules. Would you write them by hand? Copy/paste/edit?

Several limitations

- Imagine the previous example with 100 (or more) modules. Would you write them by hand? Copy/paste/edit?
- And if you want to run several tests with N modules, $N = \{1, 2, 3, 5, 10, 15, 100, 1000\}$?
- And if some of the modules are different from the others?
  $\Rightarrow$ You can't use variable renaming.
  $\Rightarrow$ Lots of code duplication. Error prone. Not flexible.

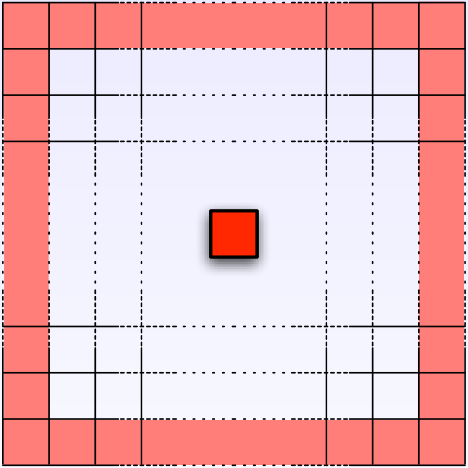
Outline

Sensor networks



The sensor in the middle broadcasts the alert.
Its code must be different.

Sensor networks



The sensors on the edges are not completely surrounded.
Their code for sensing alerts is different.

Possible solutions

- We want to model-check sensor networks with many different parameters.
- Generate PRISM code with scripts.

Possible solutions

- We want to model-check sensor networks with many different parameters.
- Generate PRISM code with scripts:
  - Use shell/M4/Ruby/Perl/Python/<You name it> scripts.
  - No real standard.

Possible solutions

- We want to model-check sensor networks with many different parameters.
- Generate PRISM code with scripts:
  - Use shell/M4/Ruby/Perl/Python/<You name it> scripts.
    $\Rightarrow$ You need to know a scripting language.
  - No real standard.

Possible solutions

- We want to model-check sensor networks with many different parameters.
- Generate PRISM code with scripts:
  - Use shell/M4/Ruby/Perl/Python/<You name it> scripts.
    $\Rightarrow$ You need to know a scripting language.
    $\Rightarrow$ Bugs in your script will be hard to debug.
  - No real standard.

Possible solutions

- We want to model-check sensor networks with many different parameters.
- Generate PRISM code with scripts:
  - Use shell/M4/Ruby/Perl/Python/<You name it> scripts.
    $\Rightarrow$ You need to know a scripting language.
    $\Rightarrow$ Bugs in your script will be hard to debug.
    $\Rightarrow$ Your attention is distracted from your first objective.
  - No real standard.

Outline

eXtended Reactive Modules

- We feel that we need an extended version of the PRISM language.

eXtended Reactive Modules

- We feel that we need an extended version of the PRISM language featuring:
  - For loops.
  - If statements.
  - Functions to factor code in common.

eXtended Reactive Modules

- We feel that we need an extended version of the PRISM language featuring:
  - For loops.
  - If statements.
  - Functions to factor code in common.
- We want some kind of compiler that generates PRISM code.

eXtended Reactive Modules

- We feel that we need an extended version of the PRISM language featuring:
  - For loops at the meta-level.
  - If statements at the meta-level.
  - Functions to factor code in common at the meta-level.
- We want some kind of compiler that generates PRISM code.
  $\Rightarrow$ Meta-programming: code partially generated and evaluated at compile time.
  $\Rightarrow$ Consistency of the generated code is ensured by the compiler.
  $\Rightarrow$ Type-checking is possible.

Outline

Using eXtended Reactive Modules

XRM's tools are built with the Stratego/XT bundle.

Using eXtended Reactive Modules

XRM's tools are built with the Stratego/XT bundle.

- Stratego: a language designed for program transformations.

Using eXtended Reactive Modules

XRM's tools are built with the Stratego/XT bundle.

- Stratego: a language designed for program transformations.
- SDF: Syntax Definition Formalism.
  Modular definitions make it easy to:
  - Extend grammars.
  - Embed a grammar into another.

Using eXtended Reactive Modules

XRM's tools are built with the Stratego/XT bundle.

- Stratego: a language designed for program transformations.
- SDF: Syntax Definition Formalism.
  Modular definitions make it easy to:
  - Extend grammars.
  - Embed a grammar into another.
- SGLR: Scannerless Generalized LR parser.
  - Enables ambiguities.
  - Provides several disambiguation filters.

Tools for working with eXtended Reactive Modules

XRM comes with several tools:

- 4 parsers.
- 4 pretty-printers.

Tools for working with eXtended Reactive Modules

XRM comes with several tools:

- 4 parsers.
  - PRISM language.
  - XRM language (extended PRISM).
- 4 pretty-printers.

Tools for working with eXtended Reactive Modules

XRM comes with several tools:

- 4 parsers.
  - PRISM language.
  - XRM language (extended PRISM).
  - PCTL language (for specifying properties to model-check).
  - XPCTL language (PCTL extended with XRM embeddings).
- 4 pretty-printers.

Tools for working with eXtended Reactive Modules

XRM comes with several tools:

- 4 parsers.
  - PRISM language.
  - XRM language (extended PRISM).
  - PCTL language (for specifying properties to model-check).
  - XPCTL language (PCTL extended with XRM embeddings).
- 4 pretty-printers.
- xrm-front: Front-end that compiles XRM (resp. XPCTL) files into standard PRISM (resp. PCTL) files.

Outline

Meta-programming: Meta-For loops (1/2)

Many of the real-world examples must be modelised with many modules. Meta-For loops are one of the most useful features of XRM when it comes to large systems.

Writing sensor networks with XRM

```
const int width = 100;
const int height = 100;

for x from 0 to width − 1 do
  for y from 0 to height − 1 do
    module sensor[x][y]
      status[x][y] : [0..MAX_STATE] init SENSE;
      // Commands of the module go here.
    endmodule
  end
end
```

Meta-programming: Meta-For loops (1/2)

Here, x and y are declared as meta-vars (variables at the meta-level, that won't exist in the resulting source code). The for loop will be unrolled by xrm-front.

Writing sensor networks with XRM

```
const int width = 100;
const int height = 100;

for x from 0 to width - 1 do
  for y from 0 to height - 1 do
    module sensor[x][y]
      status[x][y] : [0..MAX_STATE] init SENSE;
      // Commands of the module go here.
    endmodule
  end
end
```

Meta-programming: Meta-For loops (2/2)

XRM also has shell-like meta for loops.

Shell-like meta-for loop

```
module xrm
  x : [0..1]  init 0;
  y : [0..10] init 0;
  z : [0..1]  init 0;
  for i in x, 1+2, y do
    [] y=i -> y' = y+1;
  end
endmodule
```

Meta-programming: Meta-If statements

Conditional definition of a module

```
// Coordinates of the sensor broadcasting the alert.
const int event_x = 5;
const int event_y = 5;

for x from 0 to width - 1 do
  for y from 0 to height - 1 do
    module sensor[x][y]
      if x = event_x & y = event_y then
        // This node is the node broadcasting the alert.
      else
        // Other nodes are defined here.
      end
    endmodule
  end
end
```

XRM Arrays

- Large modules require many variables.

XRM Arrays

- Large modules require many variables.
- XRM enables multi-dimensional array declarations.
- Array subscripts must be evaluable down to positive integers at compile time.

XRM Arrays

- Large modules require many variables.
- XRM enables multi-dimensional array declarations.
- Array subscripts must be evaluable down to positive integers at compile time.

XRM Arrays

```
const int N = 4;
const int M = 2;
module
  // multi-dimensional "sparse" array
  x[0..10][0,2,5..7] : [0..1] init 0;
  [] x[N][M]=0 -> (x[N][M]'=1);
endmodule
```

XRM Builtins

For the time being, XRM features two new builtins for generating random variables:

XRM's builtins

```
module sample
  x : [0..51] init 0;
  [] true -> x'=static_rand(42);
  [] true -> x'=rand(42);
endmodule
```

XRM Builtins

For the time being, XRM features two new builtins for generating random variables:

Generated code

```
module sample
  x : [0..51] init 0;
  [] true -> x'=<random value>;
  [] true -> x'=__rand_0;
endmodule
module __rand_0
  __rand_0 : [0..42];
  [] true -> 1/43:(__rand_0'=0) + 1/43:(__rand_0'=1) +
             1/43:(__rand_0'=2) + ...
             ... + 1/43:(__rand_0'=42);
endmodule
```

XRM Parameterized formulas

Parameterized formulas are inlined at their call site.

Code factorized with eXtended formulas

```
const int POWER = 42;

formula consume (int value) =
  battery' = battery < value ? 0 : battery - value;
formula must_wake_up = // Some condition ;

module sensor
  battery : [0..POWER] init POWER;
  // ...
  [] must_wake_up -> 1:consume(WAKE_UP_COST);
endmodule
```

eXtended PCTL and other features

- PCTL stands for Probabilistic Computational Tree Logic. It's the language used for specifying properties to model-check.
- XPCTL = PCTL + XRM extensions.
  - Meta-code.
  - Arrays.
  - Parameterized formulas.

eXtended PCTL and other features

- PCTL stands for Probabilistic Computational Tree Logic. It's the language used for specifying properties to model-check.
- XPCTL = PCTL + XRM extensions.
  - Meta-code.
  - Arrays.
  - Parameterized formulas.
- xrm-front can perform as much partial evaluation as possible (constant propagation and constant expression evaluation).

eXtended Reactive Modules in action

- [Demaille et al., 2006]
- Implementation in Shell + M4/m4sugar:
- Implementation with eXtended Reactive Modules:

eXtended Reactive Modules in action

- [Demaille et al., 2006]
- Implementation in Shell + M4/m4sugar:

- Implementation with eXtended Reactive Modules:

eXtended Reactive Modules in action

- [Demaille et al., 2006]
- Implementation in Shell + M4/m4sugar:
  - 264 lines of M4 + 247 lines of Shell script.
- Implementation with eXtended Reactive Modules:
  - 87 lines of XRM + 12 lines of XPCTL.

eXtended Reactive Modules in action

- [Demaille et al., 2006]
- Implementation in Shell + M4/m4sugar:
  - 264 lines of M4 + 247 lines of Shell script.
  - Generates 1316 lines of PRISM + 25 lines of PCTL.
- Implementation with eXtended Reactive Modules:
  - 87 lines of XRM + 12 lines of XPCTL.
  - Generates 941 lines of PRISM + 25 lines of PCTL.

In conclusion...

- eXtended Reactive Modules provides a quite complete and reliable way of performing model-checking on large models.
- Benefit from APMC's ability to handle large systems.
- XRM is quite reliable and passes 93% of the 616 tests of its test suite.

In conclusion...

- eXtended Reactive Modules provides a quite complete and reliable way of performing model-checking on large models.
- Benefit from APMC's ability to handle large systems.
- XRM is quite reliable and passes 93% of the 616 tests of its test suite.

Future work:

- Type checking. Bound checking.
- Non-static array accesses.
- Modularity through imports.
- Optimizations.

In conclusion...

- eXtended Reactive Modules provides a quite complete and reliable way of performing model-checking on large models.
- Benefit from APMC's ability to handle large systems.
- XRM is quite reliable and passes 93% of the 616 tests of its test suite.

Future work:

- Type checking. Bound checking.
- Non-static array accesses.
- Modularity through imports.
- Optimizations.
- C Back-end to replace PRISM's compiler.

Bibliography I

- Alur, R. and Henzinger, T. A. (1999).
  Reactive modules.
  Formal Methods in System Design.

- Bravenboer, M., van Dam, A., Olmos, K., and Visser, E. (2005).
  Program transformation with scoped dynamic rewrite rules.
  Technical Report UU-CS-2005-005, Institute of Information and Computing Sciences, Utrecht University.

Bibliography II

- Demaille, A., Peyronnet, S., and Hérault, T. (2006).
  Probabilistic verification of sensor networks.
  In *Proceedings of the Fourth IEEE International Conference on Computer Sciences, Research, Innovation and Vision for the Future (RIVF)*, Ho Chi Minh City, Vietnam.

- LRDE — EPITA Research and Developpement Laboratory (2005).
  Transformers home page.
  http://transformers.lrde.epita.fr.

- Stratego.
  http://www.stratego-language.org.

Bibliography III

xrm-svn.
`https://svn.lrde.epita.fr/svn/xrm/`.