eXtended Reactive Modules

Benoit Sigoure

Technical Report n^o0620, July 2006
revision 1083

Abstract: Reactive Modules is a formal model used to represent synchronous and asynchronous components of a system. PRISM is a widely used probabilistic model-checker. It introduced the PRISM language, highly based on the Reactive Modules formalism. This language quickly reaches its limit when it comes to large models.
eXtended Reactive Modules (XRM) is an extension of the PRISM language. It comes with a compiler that translate XRM modules in PRISM modules, thus providing a comprehensive and reliable solution for people willing to write large models.

Résumé: Reactive Modules est un modèle formel utilisé pour décrire les éléments synchrones et asynchrones d'un système. PRISM est un outil de model-checking probabiliste. Il a introduit le langage PRISM, grandement basé sur le formalisme de Reactive Modules. Ce langage atteinte vite ses limites lorsqu'il s'agit de décrire des modèles conséquants.
eXtended Reactive Modules est une extension du langage PRISM. Il est fournit avec un compilateur qui traduit les modules XRM en modules PRISM, fournissant ainsi une solution fiable et complète pour les gens ayant besoin de décrire des systèmes conséquants.

Keywords
eXtended Reactive Modules, Reactive Modules, PRISM, XRM, Model checking, Stratego

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
benoit.sigoure@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Contents

Chapter 1

Introduction

Reactive Modules is a formal model used to represent synchronous and asynchronous components of a system. It has never really been used as a practical computer language but it is a formalism used to *describe* systems. PRISM is a tool widely used in probabilistic model checking. It features the PRISM language which is highly based on the Reactive Modules formalism. This language has been found to have several limitations when it comes to large systems containing tens or hundreds of modules. It does not enable to easily parameterize the model so that it can be checked with different parameters. This comes from the fact that this language was simply designed to *describe* systems, not to actually program them.

The goal of eXtended Reactive Modules is to provide a comprehensive solution to these problems by adding syntactic extensions to the PRISM language. XRM comes with a set of tools which enable the programmer to work with the extended version of the PRISM language. The main tool, `xrm-front`, will compile an XRM source file (written in the extended language) in a PRISM source file (the base language, used by the tools PRISM and APMC).

In this document, we will describe how to use the XRM package, and what are the syntactic extensions featured. We assume that the reader is comfortable with model checking, with the Reactive Modules formalism and especially with the PRISM language. If not, please read (**?**) or (PRISM's Manual).

Chapter 2

Installation

2.1 Requirements

- Stratego/XT 0.17M1 (at least revision 15278 committed Mon, 29 May 2006) because of libstratego-gpp (added in Stratego/XT 0.17M1 rev 15278).
- A C99 compiler (such as GCC).
- ATerm 2.4.2 or newer.
- SDF2-Bundle 2.3.4 or newer.
- GNU make.

2.2 Using Nix

Nix is a package management system that ensures safe and complete installation of packages. You can get Nix from (Nix) (pick up the latest *unstable* release). Download `nix-X.YYpreZZZZ` not `nixpkgs-X.YYpreZZZZ`. Once Nix is installed, use the following commands (you might need to be root depending on how you installed nix):

```
$ nix-channel --add \
http://nix.cs.uu.nl/dist/stratego/channels-v3/strategoxt-unstable
$ nix-channel --add \
http://nix.cs.uu.nl/dist/nix/channels-v3/nixpkgs-unstable
$ nix-channel --update
$ nix-env --install aterm sdf2-bundle strategoxt
```

There you are! Add the following line to your .bashrc/.zshrc:

```
[ -r /nix/etc/profile.d/nix.sh ] && source /nix/etc/profile.d/nix.sh
```

That's it!

2.3 Without Nix

Install ATerm, SDF2 Bundle and Stratego/XT from (Str) Additional install instructions can be found there.

2.4 Installing XRM

1. Uncompress the tarball: `gunzip -c xrm-version.tar.gz | tar -xf -`
2. Use the following command to setup a build-tree:

   ```
   $ cd xrm-version && mkdir _build && cd _build
   ```

3. Invoke configure to generate the Makefiles. If you use Nix, simply use '`../configure`' If you don't use Nix, use '`../configure PKG_CONFIG_PATH=<P>`' where '`<P>`' stands for the path(s) to the directory(ies) where the `.pc` files of your Stratego/XT installation were installed. eg:

   ```
   /usr/lib/pkgconfig/aterm.pc
   /usr/lib/pkgconfig/sdf2-bundle.pc
   /usr/lib/pkgconfig/stratego*.pc
   ```

4. Then simply type '`make all check`' then '`make install`'.
5. NOTE: If you see many warnings/errors from `SdfChecker` during compilation, don't worry, it is normal (unless it actually stops the build).

2.5 Tools provided with XRM

- `xrm-front`: The front-end provided by XRM will take as input an XRM source code and will transform it into a standard PRISM source code.
- `parse-prism`: Parses a PRISM source code and yield an AST in ATerms.
- `parse-xrm`, `parse-pctl`, `parse-xpctl`: Ditto with XRM/PCTL/XPCTL source code.
- `pp-prism`: Pretty prints ("unparses") a PRISM AST as PRISM source code.
- `pp-xrm`, `pp-pctl`, `pp-xpctl`: Ditto with XRM/PCTL/XPCTL source code.

Chapter 3

Using xrm-front

`xrm-front` is the main tool of the XRM package. It transforms a source written in XRM into a standard PRISM source.

3.1 Common options

The options can be reviewed by invoking `xrm-front` with the `--help` argument. Common options include:

`-i | --input`	Specify the input file.
`-o | --output`	Specify the output file.
`-D | --desugar`	Desugar the generated PRISM code.
`--verbose notice`	Keep you informed about stages of the pipeline.
`-A | --pp-aterm`	Pretty print output with `pp-aterm`.
`-p | --pctl`	The input file is an eXtended PCTL file.

3.2 Return value

`xrm-front` will return 0 if it succeeds and non zero if an error occured. Possible return values are:

- 1: rewriting failed (eg: it might be a bug in `xrm-front`)
- 2: error with meta-vars (eg: undefined meta-var, redefined meta-var)
- 3: arithmetic error when evaluating code (eg: division/modulo by 0)
- 4: invalid call to a builtin (eg: **rand**(1,2,3))
- 5: errors related with arrays (eg: subscript is not a positive integer)
- 6: invalid call to a parameterized formula (eg: not enough arguments)
- 42: internal compiler error (please send a bug report)
- 51: not yet implemented

Chapter 4

The XRM language

4.1 Foreword

The specification of the base language is documented in PRISM's manual (a copy is available in the `prism/` directory).

The XRM language is fully PRISM compliant and only offers extension to the base language.

Many people used to overcome the limitations of the PRISM language by generating PRISM code using scripts (Shell scripts, M4 scripts, etc...) (2). They came to this kind of solution because they needed to generate large systems which is almost impossible to do by hand. Indeed, most of the time, a system is made of several identical modules which are interacting with each other. Traditionally, this is done using module renaming. Nevertheless, it is still impossible to easily parameterize the number of modules in the system. Moreover when a module is slightly different than the others, it is impossible to use module renaming. Thus, PRISM files are usually highly redundant and hard to write by hand. That is why it is so common to generate PRISM code using scripts.

However, scripts are generally not aware of the PRISM code they are generating. It is easy to generate invalid code. In this case, one has to find where is the error in the script and fix it. The advantage of using a compiler such as `xrm-front` becomes clear: a compiler doesn't carry around strings, it works with an AST (Abstract Syntax Tree) generated by a parser itself guided by a grammar. Most error will be detected and reported at parse time or compile time, making the developing process more reliable.

Since the main concern is about code generation, XRM enables a form of meta-programming. For instance, the number of modules in a system could be stored in a variable and a *for* loop could be used to generate them. For loops don't exist in PRISM. In order to transform an XRM *for* loop in a standard PRISM code, we have to unroll the loop, which leads to code generation (meta-programming).

We will now see what are the syntactic extensions featured by XRM and how to use them.

4.2 XRM Modules

Within XRM modules, it is not mandatory to specify declarations before commands. They can be freely intertwined in any particular order (whereas in plain PRISM, declarations must come first and then commands will follow). The following module is valid in XRM:

```
module OutOfOrder
  [] x=0 -> (x'=1);     // command
  x : [0..1] init 0;    // declaration
endmodule
```

The last step of `xrm-front`'s pipeline is to group all declarations together at the beginning of the module so that the module will be valid standard PRISM.

4.3 XRM Expressions

1. XRM has arrays. Array accesses (or subscripted arrays, eg: x[3]) are expressions.
2. XRM introduces two operators: << and >> which have the same semantics as in C. They are desugared using calls to the builtin `pow`.
3. It is possible to disambiguate a double value from an integer by prefixing the literal value with either `d` or `D` or `f` or `F`, eg:

   ```
   const double p = 1D; // same as 1.0
   ```

   This is mainly used internally for concrete syntax.

4.4 XRM Arrays

1. Array accesses can be found (nearly) everywhere a simple variable identifier is allowed in PRISM.
2. It is possible to declare an array of variables instead of a several variables. Everywhere a variable declaration was allowed in PRISM, an array declaration is allowed in XRM.
3. Arrays are declared with ranges, eg:

   ```
   x[0..4] : [0..1] init 0;
   ```

   will declare an array of 4 elements: x[0], x[1], x[2] and x[3].
4. It is possible to declare multidimensional arrays, eg:

   ```
   x[0..3][0..4] : [0..1] init 0;
   ```

   For multidimensional arrays intermediate dimensions are not accessible (simply because they don't exist). So the former declaration will declare:

   ```
   x[0][0], x[0][1], x[0][2], x[0][3],
   x[1][0], x[1][1], x[1][2], x[1][3],
   x[2][0], x[2][1], x[2][2], x[2][3]
   ```

But in this case accessing x[0] for example doesn't make sense. If you want to access x[0] you have to declare it using a 1-dimension array. This array can coexist with x[][].

5. It is possible to use any kind of range to declare an array, eg:
   x[2..4][0,3..5] will declare only: x[2][0], x[2][3], x[2][4], x[2][5],
   x[3][0], x[3][3], x[3][4], x[3][5],
   x[4][0], x[4][3], x[4][4], x[4][5]
   The Cartesian product of all the dimensions of the array is used to compute the set of variables defined.

6. Arrays can be declared in multiple parts, eg:

   ```
   x[0..3] : bool init true;
   x[4..7] : [1..5] init 2;
   ```

   In this case, x[0] through x[3] contain boolean values whereas x[4] through x[7] contain integer values between 1 and 5 included. Note that in this case, multiple declarations must not overlap, eg:

   ```
   x[0..4] : bool init true;
   x[4..7] : [1..5] init 2; // x[4] overlaps with the previous
                            // declaration!
   ```

   This will be caught as an error by `xrm-front`.

7. Arrays can also be implicitly declared using *for* loops.

8. Update of values in arrays is illustrated in the following example:

   ```
   [] x[N][M]=0 -> (x[N][M]'=1);
   ```

   Notice that the prime (′) comes after the dimensions of the array access.

9. All the subscripts in array accesses must be evaluable down to a simple positive integer at compile time, eg:

   ```
   const int N = 3;
   // ...
   i : [0..42] init 0; // declaration of 'i'.
   [] x[N+3]=0 -> ...; // valid: N+3 can be worked out at compile time.
   [] x[i+3]=0 -> ...; // invalid: the value of 'i' is dynamic and unknown
                       // at compile time.
   ```

10. NOTE: In XRM 1.0 and before, it was possible to declare arrays like in C, eg:

    ```
    const int array[N];
    ```

    was automatically desugared to:

    ```
    const int array[0..N-1];
    ```

    This is no longer true since XRM 1.1, one must now explicitly write **const int** array [0.. N−1]. Since then, **const int** array[N] leads to the declaration of a single variable, that is, the N^{th} variable of the array.

4.5 XRM Meta-code

XRM introduces meta-*for* loops and meta-if statements in the language. These constructs are said to be 'meta' because they are evaluated by `xrm-front` and lead to code generation.

4.5.1 XRM For loops

1. Meta-*for* loops can be found in only 2 places within a XRM source file:

   (a) Where we could expect a module declaration.

   (b) Where we could expect a declaration or a command, within a module.

   This implies that meta-*for* loops can only be used to generate modules (or other file sections, such as formulas, globals, etc.) or commands and declarations within modules.

2. There exists 2 flavors of *for* loops:

   (a) *For* loops *à la* Pascal

   (b) *For* loops *à la* Shell

   Example:

   ```
   for i from 0 to 3 do ... end                   // Pascal-like
   for i from 0 to 10 step 2 do ... end           // Pascal-like
   for i in a, 1+2, N do .. end                   // Shell-like
   ```

   In each of these 3 cases, the variable *i* will be considered as a meta-variable, meaning it will only exists at the meta-level and won't appear in the final source code.

3. The word `for` is a reserved keyword in XRM and cannot be used for an identifier.

4. For Pascal-like for-loops, the fields *from*, *to* and *step* must be evaluable down to simple integers at compile time. The value of the field *from* must be less than or equal to that of the field *to*.

5. For-loops are unrolled by copying the body of the loop and replacing every match of the identifier of the meta-var by its successive values.

6. For-loops are the only way of declaring a meta-var at the moment.

7. For-loops can be used to create new modules. Since each module must have a unique name, it will have to be suffixed by an array access using a meta-var, eg:

   ```
   for i from 1 to 3 do
     module dummy[i]
       x[i] : [0..1] init 0;
     endmodule
   end
   ```

   will generate 3 modules: dummy[1], dummy[2] and dummy[3]. Thus in this case you get 3 different modules, each with their own single unique variable *x*. If you wish to provide each module with an array of, say 6 elements, then do the following:

```
for i from 1 to 3 do
  module dummy[i]
    x[i][0..5] : [0..1] init 0;
  endmodule
end
```

In this case the first dimensions (*i*) will first be expansed by the loop unrolling, which will equip each generated module with its own unique variable *x* and then the second dimension will create an array of 6 elements for each generated module.

8. Meta-*for* loops can also be used to declare arrays implicitly:

```
module ImplicitArray
  for i from 0 to 3 do
    x[i] : [0..i] init i;
  end
endmodule
```

In this case the module ImplicitArray will have a single array named *x* of 4 elements. This method offers a greater control on how each element of the array is declared.

4.5.2 XRM Meta-If Statements

1. Meta-if statements can be found in only 3 places within a XRM source file:
   (a) Where we could expect a module declaration.
   (b) Where we could expect a declaration or a command, within a module.
   (c) Where we would expect an expression.

   However the latter case has a restriction that the two formers don't have: the then-part and the else-part of the if statements cannot contain more than one expression.

2. The syntax for meta-if statements is illustrated in the following example:

```
if true then
  module alwaysGenerated
    if 0 = 42 - 21 then
      neverGenerated : [0..42] init 0;
    else
      alwaysGeneratedToo : [0..42] init 0;
    end
  endmodule
end
```

3. The condition of the meta-if statements must be evaluable at compile time. It must either be evaluable down to true or false, or down to a simple integer/double. If that integer/double is 0, the condition will be false, otherwise it will be true (like in C).

4. When a comparison on reals occur in the XRM-compiler, they are carried out with a precision of 10^{-7} which means that if the condition is reduced down to 0.00000001 (for instance) it will be evaluated as being **false** because this value is 0 when used with a precision of 10^{-7}.

4.6 XRM builtins

XRM introduces two new builtins for generating random numbers: `rand` and `static_rand`.

1. In XRM, `rand` and `static_rand` are reserved keywords and cannot be used for identifiers.
2. Both `rand` and `static_rand` take either one or two arguments which must be evaluable down to simple integers at compile-time. If `rand` or `static_rand` is called with a single argument, a second argument (an integer equal to zero) is added. If the single argument is positive, the zero is added before it, otherwise it is added after it, eg:

   ```
   rand(3)  // will be desugared to rand(0, 3)
   rand(−3) // will be desugared to rand(−3, 0)
   ```

3. **static_rand**(low, hi) will be transformed into a random integer ranging from 'low' to 'hi' (included). The random number is obtain with `rand(3)` which is seeded with the current UNIX time-stamp when `xrm-front` starts.
4. NOTE: It is possible to specify a seed with `xrm-front`'s `-s | --seed` option.
5. Calls to `static_rand` will be evaluated after unrolling of meta-for loops to ensure that each iteration of the loop gets its own random number. It will be evaluated earlier if it happens to be used where a statically evaluable value is required more early in the pipeline (eg: in meta-if statements' condition, in the fields 'from', 'to' or 'step' of a meta-*for* loop, etc.)
6. Bear in mind that the random numbers generated by `static_rand` are constant from one run to another unless you re-generate the PRISM source (with `xrm-front`) each time before running.
7. **rand**(low, hi) will be transformed into a new variable (each call to `rand` will generate a new unique variable) which will be controlled by an external module with a single command:

   ```
   module testRand                        module testRand
     x : [0..42] init 0;                    x : [0..42] init 0;
     [] x=0 -> x'=rand(42);                 [] x=0 -> x'=__rand_0;
   endmodule                              endmodule
                                          module __rand_0
                                   =>        __rand_0 : [0..42];
                                             [] true -> 1/43:(__rand_0'=0)
                                                      + 1/43:(__rand_0'=1)
                                                       ...
                                                      + 1/43:(__rand_0'=42);
                                          endmodule
   ```

   This is not a reliable random number generator! Depending on the type of model used (CTMC, DTMC, MDP) the modules will not always be "scheduled" one after the other. For instance, the module `testRand` could run twice in a row and thus use twice the same "random" value.

8. TODO: The current implementation of `rand` will be renamed (maybe as `bad_rand` or `old_rand`) and a new reliable implementation will be provided as a replacement. In this implementation, the random variable won't be hosted in a foreign module anymore, it will be hosted directly in the module which called `rand`. The variable will be updated each time it is accessed to ensure a real and constant randomness of the numbers generated.

4.7 XRM Parameterized formulas

XRM formulas have been eXtended and can now be parameterized, eg:

```
formula isfree(int i) = p[i]=0..4,6,10;
```

1. Parameterized formulas can have 4 kinds of arguments:
   (a) int (as in the example above)
   (b) double
   (c) bool
   (d) exp
2. With the `exp` type, the formula behaves a bit like C's macro-functions. We can also see that as a "catch-all" type since no type-checking will be performed on this kind of arguments.
3. Parameterized formulas definitions will be removed in the output PRISM code. Invoking parameterized formulas is somewhat like calling a function, eg: isfree (2) will be inlined as p [2]=0..4,6,10;
4. Note that using the PRISM-3 calling style for builtins cannot be used for formulas, eg: **func**(isfree , 2) is not supported at the moment. We *might* add support for this.
5. Once a parameterized formula has been defined, it can't be redefined/undefined, just like a normal PRISM formula.
6. Parameterized formula can also define updates, not only expressions, eg:

```
formula consume (int value) =
  battery' = battery < value ? 0 : battery - value;
// ...
module sensor
  battery : [0..POWER] init POWER;
  // ...
  [] must_wake_up -> 1:consume(WAKE_UP_COST);
endmodule
```

4.8 XRM Keywords

XRM uses the following additional reserved keywords:
`for`, `rand`, `static_rand`, `func`
They cannot be used as identifiers.

Chapter 5

XRM and Property Files

Almost all the features of the XRM language are available in eXtended Property Files. There is an exception:

1. The rand builtin cannot be used in property files.

It is also possible to define formulas and parameterized formulas in XPCTL files.

If you wish, it is possible to specify properties directly in XRM files. You can add property sections to your XRM files. Property sections can be found everywhere a module declaration can be found. A property section is specified as:

```
properties
  // XPCTL code here
end
```

If you use property sections, you will need to pass an additional argument to `xrm-front` to specify the PCTL property file where the properties must be saved. The option switch is `--p-output f` or `-po f` (where `'f'` is a path). However, this option is not mandatory. If you specify property sections but omit this switch, `xrm-front` will discard the properties and issue a warning about that.

Chapter 6

Forthcoming features and known bugs

6.1 Forthcoming features

The following features are not yet implemented (or only partially implemented or broken). They are ordered in term of the estimated time needed to successfully implement them. For a complete list of things to be done please review the `TODO` file. You can also review the `trac` located at (xrm)

1. More sanity checks for all declarations at different stages of the pipeline to ensure that everything is well defined. (variables used have been declared somewhere etc.)
2. Possibility to import another module, eg: import common.pm
3. Parameterized formulas. (Pretty much like macro-functions in C)
4. Scopes for meta variables (allow redefinitions/shadowing).
5. Bound/Type checking (ensure that variables are properly used according to their type/domain definition).
6. Conditional tests on arrays. `?=` and `?!=` operators for arrays. eg:

```
x[1..3]=0          => x_1=0 & x_2=0 & x_3=0
x[1..3]?=0         => x_1=0 | x_2=0 | x_3=0
```

7. Dynamic array accesses, eg: x[i] where `i` is not known at compile-time.
8. Array initializations *à la* C:

```
x[0..3] : [0..4] init {0, 1, 2};
const int array[0..3] = {0, 1, 2};
```

9. Better error messages (with location of the error).

6.2 Known bugs

1. The construct **system** ... **endsystem** (for system compositions) is broken at the moment (meaning: using it will result in a parse error). It is probably a simple problem with the base SDF grammar. I've never seen a PRISM source using this construct so this has been in the `TODO` list since the beginning but with a very low priority.

2. Unary operators are allowed in the base language whereas they should not (because they are not allowed in the original PRISM parser).

3. There is nearly no warranty that the generated code will work in PRISM. Generally speaking, if the input XRM source is correct, the output PRISM source will also be correct. However, at this stage of the development, the front-end is still pretty fragile and I am sure it is quite easy to generate invalid PRISM code without having any error reported by `xrm-front` (in this case please report the bug). What we clearly need to thwart this is:

   (a) Type checking.

   (b) Bound checking.

4. In Property Files, paths using "bounded until" can lead to ambiguities in XPCTL. Eg:

   ```
   Pmin=? [ true U<=k (1)+1=1 ]
   ```

   can be parsed as:

   ```
   Pmin=? [ true U<=(k(1)) (+1)=1 ]
   // where k(1) is a call to a parameterized formula.
   ```

   -or-

   ```
   Pmin=? [ true U<=(k) (1+1)=1 ]
   // which is most likely the intended meaning.
   ```

   In this case, `parse-xpctl` will fail with an error that is like the following:

   ```
   sglr:error: Ambiguity in your-file.pctl, line L, col C:
   Formula "U" "<=" Expression Formula -> Path {cons("BoundedUntilLtEq'
   Formula "U" "<=" Expression Formula -> Path {cons("BoundedUntilLtEq'
   Formula "U" "<=" Expression Formula -> Path {cons("BoundedUntilLtEq'
   ./src/tools/parse-xpctl: rewriting failed
   ```

   In this case, add a couple of parenthesis to remove the ambiguity:

   ```
   Pmin=? [ true U<=(k) (1)+1=1 ]
   ```

5. Probably many other things (run `make check` and see the tests that fail).

For a complete list of bugs, you should review the `trac` located at (xrm).

Chapter 7

Internal Documentation

7.1 Introduction

This part of the documentation aims at explaining the internals of XRM, how it has been developed and how it works. This is only useful if you intend to further develop XRM or to understand the underlying code. This could also be useful if you're learning Stratego.

7.2 General design

This sections explains the general design used when developing XRM. The sources are located under the `src/` folder. They are divided in 5 sections:

1. `src/lib`: Contains strategies/code exported in libraries linked with the tools. In fact (at this time) XRM doesn't use any library but the code under this path should be exported through shared and static libraries.

2. `src/lib/native`: Contains C code used by Stratego code (with `prim`). Actually this code features several extensions used to manipulate float values in Stratego. This was implemented in C because it wasn't available in `stratego-lib` and because doing the same work would be a pain in C. This code also provide a random number generator.

3. `src/lib/<lang>/pp`: Where `<lang>` is {pctl,prism,xpctl,xrm}. Contain the pretty-printing strategies used by the pretty printers. These strategies are meant to be available in libraries but at this time they are simply imported by the pretty-printers. Exporting them in libraries would reduce compilation time. They should rather be exported in static libraries (if possible) because dynamic libraries have inconvenient: the `.so` must be installed and available in the `LD_LIBRARY_PATH` (or must be installed in standard paths). Installing one dynamic library per pretty-printer does not seem quite attractive.

4. `src/sig`: Contains nothing. This is where signatures for the 4 languages (pctl, prism, xpctl, xrm) will be generated (under the build dir).

5. `src/str`: Contains the Stratego code used by `xrm-front`.

6. `src/syn/<lang>`: Contains the SDF grammar of the language `<lang>`. Each grammar has Stratego embeddings (for concrete syntax) defined in `Stratego<LANG>.sdf` and

optionally `<LANG>-MetaCongruences.sdf` and `<LANG>-MetaVars.sdf`. Renamed version of each grammar are generated during the build so that the grammars can easily be assimilated within other host languages later on.

7. `src/tools`: Contains Stratego sources for the pretty printers and parsers (which are registered as XTC components).

7.2.1 The build system

XRM use the autotools to set up the build system. It also uses the Makefile provided by (autoxt) and Transformers' (3) Makefile. The `configure` script tries to guess the correct value for `PKG_CONFIG_PATH` if it is not provided by looking for common locations where Stratego/XT is installed by Nix. Many GNU make extensions are used by the Makefiles. Parallel builds work and are encouraged.

7.2.2 Creating tools with XTC

XRM uses XTC-registered components. For the documentation of XTC see the chapters 28 (XTC) and 29 (Library Building) of the Stratego/XT manual. Basically, XTC components are simply applications or files which are registered by a program called `xtc`. This program creates a file where registered components are listed (the repository).

When a component is registered in the repository, its name, version and path are saved. This is useful so that you can develop tiny applications and call them one after the other in a pipeline. Calling them from Stratego is made easy because the Stratego programs know where the XTC repository is and it can query that repository to find the auxiliary tools it needs.

For instance one could register `ls`, `grep` and `wc` in an XTC repository. Then one could invoke them from a Stratego program one after the other in order to simulate the command '`ls -l | grep rwx | wc -l`' for instance. Note that XTC repositories are in fact ATerms and you can display them with `pp-aterm`.

This is used by parsers which need to invoke `sglr` with the correct parse table in argument. The parse table is generated somewhere and is registered in the XTC repository. The parser looks up the location of the parse table through the repository and then invokes `sglr` with the right path to the parse table.

However, this has a disadvantage: invoking XTC components is quite inefficient. This is because what happens actually is that, the current term is saved in a temporary file under /tmp. When the current term is quite large this can produce files from several hundred mega-bytes up to several giga-bytes. Then the XTC component you invoked is called with that temporary file as input (`-i`) and another new temporary file as output (`-o`). Once it has finished, the current process reloads its current term from the temporary file where the XTC component sent its output.

This leads to many i/o operations on disk which tend to be quite slow. That is why XRM tries to avoid to use XTC for external processes as much as possible. But since we still want to be modular, instead of using XTC components, we use external libraries. At this time XRM doesn't use libraries because it was easier to import the libraries required than to export them through libraries which would then be linked with the binary. However this makes the compilation time longer. We should really consider exporting common things in external libraries.

The `--help` and `--about` options are handled using (tool-doc). The pretty-printers directly include the strategies they need (the pretty-printing rules are written in Stratego under `src/lib/<lang>/pp`). This enables them to be stand-alone. They can pretty-print without using intermediate files. They also use `libstratego-gpp` which is quite recent. This enables them to use `abox-to-text` without having to call an external XTC component and go through intermediate temporary files.

7.3 `xrm-front`'s pipeline

In this section we will review the pipeline of `xrm-front`. We will explain the different stages of the pipeline, how the dynamic rules are used and how the transformations are performed.

Everything begins in `src/str/xrm-front.str`. The first thing performed by `xrm-front` is to check whether the options it was invoked with are consistent. For instance, if the user invokes `xrm-front` with the flags `-b` (request output in binary ATerm format) and `-P` (request output in pretty-printed ATerm format) the switch `-P` is ignored and a warning is issued. In former version of `xrm-front`, there were more conflicting options.

Then `xrm-front` parses its input files with the correct parser. When the switch `-p` (or `--pctl`) is provided, XRM must parse XPCTL source code whereas it parses XRM source code by default. The parsing is performed by an external XTC component. It would be useful to use library-based parser instead to improve performances. Then the strategy `xrm-front`-pipeline is invoked.

This where the transformations begin. The real pipeline can be found in `src/str/xrm-to-prism.str`.

7.3.1 First stage: remove the XRM sugar

The first stage removes the XRM sugar. This is a simple innermost traversal where basic transformations rules are applied to remove what is merely simple syntactic sugar.

We also use this stage to catch calls to the XRM builtin `rand` in property files (which is not allowed). In XRM files, these calls are simply replaced by a variable which will be the random variable. At this time each random variable is controlled by a separate module generated by `xrm-front`. This module is stored in a DR named `RandGenModules`.

7.3.2 Second stage: collect various declarations

The second stage collects static const declarations, formulas and parameterized formulas.

For property files, the formulas are removed when they are collected (because they are not allowed in standard property files).

Parameterized formulas are always removed once they are collected because they do not exist in the base languages. These declarations are collected in dynamic rules as follows:

1. `ExpandStaticConsts: id -> value`
2. `ExpandFormulas: id -> value`
3. `ExpandPFormulas: PFormulaCall(name, a*) -> e`
   this DR rewrites a call to the parameterized formula `name` (invoked with the arguments `a*`) to `e` which is the inlined version of the formula `name` with its formal parameters replaced by the parameters provided in `a*`.

7.3.3 Third stage: check meta-vars

The third stage of `xrm-front` is to check that meta-vars are used correctly. The whole code is traversed using a hand-crafted traversal that collects the declarations of meta-vars (using the appropriate scopes).

For instance, when the traversal enters a *for* loop, it registers the meta-var used to iterate in the loop in a scoped DR. Several things must be evaluable down to simple integers at compile time (such as array subscripts or the `from` or `to` of a *for* loop for instance. Indeed if you can't evaluate the latter at compile time, how can you unroll the loop if you don't know how many iterations it has to go through?).

Once we know what are the meta-vars (as well as static consts, formulas etc.), we can easily check whether an expression is evaluable as a literal value at compile time. If the expression contains only literals and identifiers known to be evaluable at compile time (such as static consts and meta-vars for instance) then the expression itself is evaluable at compile time.

7.3.4 Fourth stage: evaluate meta-code

Then an important stage starts: the meta-code is evaluated (leading to code generation). Once again, this stage uses a hand-made traversal and features: evaluation of static ifs, lazy evaluation of operator '&' and operator '`|`' (so that the user can rely on them to prevent invalid code from being evaluated, eg: x > 0 & a[x] to prevent an invalid array access).

For loops are unrolled. Calls to parameterized are inlined. When a call to a parameterized formula is inlined, we must re-start the stage on the code inlined.

7.3.5 Fifth stage: desugar array declarations

The fifth stage consists of removal of array declarations, evaluation of calls to the XRM builtin `static_rand` and collection of non-array local variable declarations. This is 3 different things but they are combined together to reduce the number of traversals.

Array declarations are rewritten as declarations of lists of variables. `xrm-front` ensures that array declarations are not overlapping with each other. Each array declaration is recorded in the DR `DeclaredArrays` which maps an `Identifier(idf) -> aa-list` where `aa-list` is the list of array subscript declared for that array.

When an array is declared in multiple parts, each part is added in the same entry of `DeclaredArrays` (we first fetch the parts previously declared, check that they don't overlap with the parts being declared, and if they don't, we concatenate them and save them back in the DR).

This part of the code might be a bit hard to read because an abstract factory is used to build declaration lists resulting of array declarations and this is a bit hard to follow.

Hopefully the numerous comments in the file will help the reader to understand how the rewriting is performed.

Non-array declarations are also collected in a DR named `DeclaredIdentifers` which maps an identifier to some information about it (such as its type, its definition range, its initial value).

These information are gathered for the type-checking stage. Note that type information of variables declared in arrays are not yet gathered.

7.3.6 Sixth stage: AST normalisation

So now, array declarations have been rewritten as declaration lists. This introduces several unwanted nested lists in the AST which are then removed using `flatten-list`.

7.3.7 Seventh stage: type-checking

Then comes the type-checking stage. At this time, this stage is quite basic and only checks that array subscripts do not lead to out of bound array accesses. We can easily do this thanks to the DR `DeclaredArrays` which maps an identifier to the dimensions declared for that array. The stage also ensures that all array accesses are made on declared arrays.

7.3.8 Eighth stage: Add generated modules

Then, we paste the module generated by the calls to the `rand` builtin (which were saved in the DR `RandGenModules`) at the end of the file.

7.3.9 Ninth stage: Remove array accesses

Now we can remove all array accesses. This is done by flattening them, eg x[1][2][3] is rewritten as x_1_2_3.

For property files, we also use this stage to expand all non-parameterized formulas since their expansions were not forced by any previous stage. We must force the expansion of non-parameterized formulas in property files because they are not allowed in the base language.

7.3.10 Tenth stage: Re-order the content of the modules

In the end, we can now re-order the content of the modules. Indeed, in XRM we allow the declarations and commands to be freely intertwined whereas in PRISM declarations must always come before the commands. This is done using a scoped DR. We traverse all the modules and collect declarations and commands in DRs (respectively `CommandList` and `DeclarationList`). Once we have them all we can easily re-construct the module with the declarations first, followed by the commands.

The resulting AST is further desugared if the `-D | --desugar` switch was provided, then it is pretty-printed in the format required by the command line (default: PRISM code).

Chapter 8

Conclusion

XRM succeeded in providing a comprehensive solution to the problems raised by large models. Its use has been demonstrated in (1) where we implement a sensors network in XRM and compare the implementation with that proposed in (2), which was based on Shell and M4/m4sugar. The XRM implementation requires 105 lines of XRM code whereas 1316 lines were needed for the Shell/M4/m4sugar implementation.

Thanks to the work performed by `xrm-front`, the developing process is made more reliable and efficient.

The Stratego/XT bundles provided an ideal framework to design XRM in a modular and extensible fashion.

Chapter 9

Bibliography

[Str] Stratego/xt continuous distribution, http://www.stratego-language.org/Stratego/ContinuousDistribution.

[Nix] Trace – transparent configuration environments, http://www.cs.uu.nl/wiki/Trace/Nix.

[xrm] Xrm's trac, http://xrm.lrde.org/.

[1] (2006). *Modeling of Sensor Networks Using XRM*.

[autoxt] autoxt. Stratego/xt – programs and tools – autoxt, http://nix.cs.uu.nl/dist/stratego/strategoxt-manual-unstable-latest/manual/chunk-chapter/ref-autoxt.html.

[2] 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.

[Library Building] Library Building. Stratego/xt manual – library building, http://nix.cs.uu.nl/dist/stratego/strategoxt-manual-unstable-latest/manual/chunk-chapter/library-building.html.

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

[PRISM's Manual] PRISM's Manual. http://www.cs.bham.ac.uk/~dxp/prism/manual/.

[stratego] stratego. http://www.stratego-language.org.

[tool-doc] tool-doc. Stratego/xt sources – tool doc, https://svn.cs.uu.nl:12443/repos/StrategoXT/strategoxt/trunk/stratego-regular/xtc/tool-doc.str.

[xrm-svn] xrm-svn. https://svn.lrde.epita.fr/svn/xrm/.

[XTC] XTC. Stratego/xt manual – xtc, http://nix.cs.uu.nl/dist/stratego/strategoxt-manual-unstable-latest/manual/chunk-chapter/xtc.html.