EPMC: the Extensible Probabilistic Model Checker

Andrea Turrini

State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences

Institute of Intelligent Software, Guangzhou

`http://iscasmc.ios.ac.cn`

EPMC

- mainly written in Java, with some parts written in C
- modular approach to perform model checking
- targets at using proven techniques from software engineering
  e.g. appropriate use of patterns (builder, delegate, etc.)
- divided into core parts and plugins (division not completely fixed)
- uses Maven, Ant, and make for the build process
- Eclipse should be developed for development
- uses external component where appropriate
- free available on GitHub
  `https://github.com/ISCAS-PMC/ePMC`
- open source, released under GPLv3

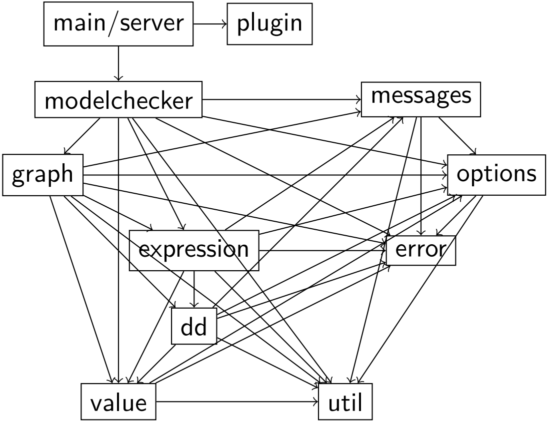
External components used

- in core part:
  - Google Guava
  - Trove
  - Java Native Access
  - JavaCC
  - Java JSON library
- additionally in one or more plugins:
  - SPOT (LTL model checking)
  - LPSolve (multi-objective model checking, IMDP lumping)
  - BDD libraries (BeeDeeDee, BuDDy, CacBDD, CUDD, JDD, Sylvan) BDD-based symbolic model checking
  - iSat3, Z3, other SMT solvers by SMTLIB interface (parametric model checking)
  - Java Algebra System/CoCoA, GiNaC (parametric model checking)
  - GMP/MPFR (arbitrary-precision model checking; experimental)
- trying to keep number of dependencies low
- new dependencies should be encapsuled in according plugin

Core part

Core part divided into components
Division between core part and plugins not yet completely fixed, some of them might be transformed to plugin

- `util`
- `main/server`
- `modelchecker`
- `graph`
- `expression`
- `value`
- `options`
- `messages`
- `error`
- `dd`
- `plugin`



util component

- contains classes and static methods not found in Google Guava
- utility classes
  - Stop watches
  - bit sets (more flexible ones)
  - bit-valued stack
  - integer deque
  - ordered maps/sets
  - bit-stream classes
  - ...
- static methods
  - instantiation helpers
  - get manifest entries
  - print stack trace
  - resource to string
  - ...

main/server components

- sets up model checking process
- can start model checking process in new process
  to prevent bugs crash the whole program
  (important for web-based usage)

modelchecker component

- `ModelChecker` coordinates model checking process
- model checking uses certain `Engines`
  - `EngineExplicit`
  - `EngineDD`
  - ...
- maps expression types to according solvers for given engine
- activates properties necessary for checking given properties
  e.g. according reward structures
- `Model` interface
  - represents models of given type, e.g. PRISM, JANI, etc.
  - load model from file
  - parse properties
  - add external properties
  - create low-level model for given engine

graph component

- representation of graphs for multiple purposes
  e.g. for low-level model or for automata
- explicit-state and BDD-based representations
- interfaces for graphs, node- and edge properties
- predefined graphs e.g. using sparse matrix structure
- predefined node and edge property classes
- state-space explorer to turn high-level model to graph
- conversion from BDD-based graphs to explicit-state graphs
- other auxiliary classes

expression component

- maintaining expressions, such as $(a = 2) \wedge \neg(b = c)$, $\mathrm{P{\scriptsize MAX}}{=}?(\mathrm{F}(a \mathrm{U} b))$
- expressions are immutable
- used both for models (e.g. on guards) and properties
- contains basic expression types
  e.g. application of operators (e.g. "$+$", "$-$", "$\wedge$", etc.)
- but less general expression types can be added by plugins
- fast evaluation of expressions
- simplification of expressions
- transformation to BDDs

value component

- type system
- Type objects used to generate new `Value`
  e.g. `TypeDouble.newValue()` creates new `ValueDouble`
- allows for typing expressions
- expression literals contain a single immutable `Value`
- for performance, `Values` are *modifiable*
- `Operator` classes to combine values
  particularly used by `expression` package
- `Value` may implement methods for addition, subtraction, etc.
- simple replacement of e.g. probability types allows to support
  - arbitrary/infinite precision computations
  - interval Markov chains or IMDPs
  - parametric models
  - quantum Markov chains
  - ...

  while reusing most parts of existing code
- for performance, specialisation is necessary
  e.g. evaluate expressions using Java `doubles`

options component

- management of options, e.g.
  engine to use, BDD package to use
- “trivial”, but important component
- quite a lot of options now
- must integrate with plugin system
- due to huge number of options, hierarchical view necessary
- blocks invalid values for options
- descriptions are stored in resource files
  - eases correction of spelling mistakes etc.
  - improves code readibility
  - eases later internationalisation

messages component

- manages output of information to users
- message: identifier plus arguments
- user readable message text stored in resource files
- better than direcly using `System.out`
- non-translated output easier to parse by other tools
- allows later internationalisation

error component

- manages problems
- uses single exception type EPMCException (inspired by PRISM)
- identifier to get information about specific problem
  e.g. parsing error, probabilities larger than one, etc.
- can be annotated by position information
  (problem caused by construct in which file, line, column?)
- user readable description stored in resource file
- non-translated output easier to parse by other tools
- allows later internationalisation

dd component

- management of decision diagrams like BDDs and MTBDDs
- relies on external BDD packages, each specified by a plugin
  e.g. BeeDeeDee, BuDDy, CacBDD, CUDD, JDD, Sylvan
- wrapper accessing these packages in a uniform way
- supports working with BDD representations of different variable types
- supports symbolically applying `Operators` on such BDDs
  for translating expressions to (MT)BDDs
- auxiliary functions not present in all packages
  e.g. enumeration of satisfying assignments
  arbitrary *n*-ary operations
- conversion of BDDs to Java memory
  faster in some cases

plugin component

- responsible for loading plugins
- can either be JAR files or directories
- defined interfaces to call functions at specific points of time (“hooks”)
    - after a command has been executed
    - before a model has been parsed
    - after a model has been parsed
    - after program options have been created

Plugins

- loaded by `plugin` component
- serve a number of different purposes
- commands: `check`, `explore`, `expression2automaton`, `help`, `lump`
- BDD packages: `beedeedee`, `buddy`, `cacbdd`, `cudd`, `jdd`, `sylvan`
- property solvers: `propositional`, `operator`, `pctl`, `ltl-lazy`, `filter`, `reward`, `coalition`, `multiobjective`
- automata: `automata`, `determinisation`
- constraint solving: `lp-solve`, `isat3`, `smt-lib`
- graph-based solvers: `graphsolver`, `graphsolver-iterative`, `graphsolver-lp`
- high-level model description languages: `jani-model`, `prism`, `rddl`
- exporters: `jani-exporter`, `prism-exporter`
- special semantical model types: `qmc`, `imdp`, `param`, `timedautomata`
- high/arbitrary precision model checking: `mpfr`, `gmp`
- hiding not strictly necessary options for tool evaluation: `specialise-qmc`, `specialise-smg`

Property solver plugins

- available solver classes are stored in a field of the `Options`
- using hook, property solver plugins add according class after options creation
- given expression to be checked
  `ModelChecker` creates instance of available solvers for given `Engine`
  calls `bool canHandle()` of this instance
  if `true`, calls `solve()`

Other plugin types

- lists of candidate classes also used for other means, e.g.
- `Operators`
- commands
- graph solvers
- high-level model types
- constraint solvers

Build process for distribution

- distinction between building for development and building for distribution because requirements are very different
- for distribution, use Maven plus Ant and some shell scripts
- packs and optimises EPMC into one JAR
- can also add required plugins
- also supports building multi-platform JAR files using cross compilers
- build time dependent on the number of plugins, but not that relevant for usage

Development using Eclipse

- for development, Eclipse should be used
- chosen because most widespread JAVA IDE, and one of the best
- build for development uses internal Eclipse building tools
- does *not* rely on Maven
- multi-platform support not needed for this task
- build time very important, to avoid annoyance during programming
- note: make sure all plugin projects are opened as well
  such that changes (renaming etc.) are propagated to all of them

The Modelling Language

- EPMC supports two different modeling languages
- JANI: JSON-based; humans-readable (kind of) but intended to be automatically processed
- PRISM modeling language: human readable

PRISM

- PRISM: most widely used probabilistic model checker
- input language: extension of Dijkstra's guarded commands

```
module two_chains
  m : [0..3];
  x : int;

  [a] m=0 -> 1.0: (x'=1000) & (m'=1);
  [b] m=0 -> 1.0: (x'=2) & (m'=1);
  [c] m=1 & x>0 -> 0.3: (x'=x-1) + 0.7: (m'=3);
  [d] m=1 & x<=0 -> 1.0: (m'=2);
endmodule

init
  m = 0 & x = 0
endinit
```

Comparison PRISM-JANI

PRISM	JANI
text-based	JSON-based
human-readable	hardly readable
extensions not that easy	easily extensible
used in PRISM, Ymer, etc.	used in EPMC, Modest toolchain, Prophesy, STORM, ...
CSP-style synchronisation	synchronisation vectors
guarded commands	locations and guards
dtmc, ctmc, mdp, pta (+smg)	lts, dtmc, ctmc, mdp, ctmdp, ma, ta, pta, sta, ha, pha, sha
been around for quite a while	published as tool paper in 2017

JANI - JSON Automata Network Interface

- joint effort between Twente, ISCAS, Aachen, and Córdoba
- intended for automatic processing
- in principle human readable
  but not intended to be done so (mainly for debugging)
- two parts:
  model description language
  tool interaction specification (skipped here)
- intended to be easy to parse and write
- intended to be easily extensible by new features
- based on JSON

EPMC

- mainly written in Java, with some parts written in C
- modular approach to perform model checking
- targets at using proven techniques from software engineering
  e.g. appropriate use of patterns (builder, delegate, etc.)
- divided into core parts and plugins (division not completely fixed)
- uses Maven, Ant, and make for the build process
- Eclipse should be developed for development
- uses external component where appropriate
- free available on GitHub
  `https://github.com/ISCAS-PMC/ePMC`
- open source, released under GPLv3

JSON - JavaScript Object Notation

- human-readable format which stores data as value-object pairs
- definition EBNF

$$
\begin{aligned}
\texttt{Json} &::= \texttt{Number} \mid \texttt{String} \mid \texttt{Boolean} \mid \texttt{Array} \mid \texttt{Object} \mid \texttt{null} \\
\texttt{Number} &::= \text{integer or real number} \\
\texttt{String} &::= \text{`"` text `"`} \\
\texttt{Boolean} &::= \texttt{true} \mid \texttt{false} \\
\texttt{Array} &::= \text{`[` `]`} \mid \text{`[`}\texttt{Json}(,\texttt{Json})^{*}\text{`]`} \\
\texttt{Object} &::= \{(\texttt{String} : \texttt{Json})^{*}\}
\end{aligned}
$$

- example

```
{"isAlive": true,
  "age": 25,
  "phoneNumbers": [
    { "type": "home", "number": "212 555-1234" },
    { "type": "office", "number": "646 555-4567"}
  ],
  "spouse": null}
```

- document types can be described in js-schema
  https://github.com/molnarg/js-schema

JANI model description

- JANI model descriptions consist of (incomplete list)
- model type (`dtmc`, `ctmc`, `mdp`, ...)
- list of actions
- list of global variables
- initial values description
- automata description
  local variables
  initial values description system behaviour
- system composition description
- properties description

js-scheme JANI model description

```
var Model = schema({
  "jani-version": Number.min(1).step(1),
  "name": String,
  "?metadata": Metadata,
  "type": ModelType,
  "?features": Array.of(ModelFeature),
  "?actions": Array.of({
    "name": Identifier,
    "?comment": String
  }),
  "?constants": Array.of(ConstantDeclaration),
  "?variables": Array.of(VariableDeclaration),
  "?restrict-initial": {
    "exp": Expression,
    "?comment": String
  },
  "automata": Array.of(Automaton),
  "system": Composition
  "?properties": Array.of(Property),
});
```

JANI version

- `jani-version` is an integer version number
- currently 1
- initial tool paper published with this version in TACAS'17
- and then started working on version 2

Model name

- `name` is the name of the model
- not interpreted usually
- thus arbitrary

Metadata

- `metadata` contain further information about the model

```
var Metadata = schema({
  "?version": String,
  "?author": String,
  "?description": String,
  "?doi": String,
  "?url": String,
});
```

- `version`: information about the version of this model (e.g. the date when it was last modified)
- `author`: information about the creator of the model
- `description`: a description of the model
- `doi`: the DOI of the paper where this model was introduced/used/described
- `url`: a URL pointing to more information about the model

Model type

- type specifies the semantics type of the model
- list of official model types below
- note: obviously, not all supporting JANI support all of them
- inofficial types can be specified using a name prefixed with "x-"

```
var ModelType = schema([
  "lts",   // labelled transition system
           // (Kripke structure, finite state automaton)
  "dtmc",  // discrete-time Markov chain
  "ctmc",  // continuous-time Markov chain
  "mdp",   // discrete-time Markov decision process
  "ctmdp", // continuous-time Markov decision process
  "ma",    // Markov automaton
  "ta",    // timed automaton
  "pta",   // probabilistic timed automaton
  "sta",   // stochastic timed automaton
  "ha",    // hybrid automaton
  "pha",   // probabilistic hybrid automaton
  "sha"    // stochastic hybrid automaton
]);
```

Additional model features

- `features` are extensions which are not bound to a particular model type

```
var ModelFeature = schema([
  "derived-operators",
  "nondet-selection",
  "arrays",
  "datatypes",
  "functions",
  "trigonometric-functions",
  "hyperbolic-functions"
]);
```

- further extensions to be specified
- feature names starting with "x-" will not be defined and are available for internal use

Actions

- are used to synchronise several parts of a model
- have a `name` and an informal description

```
"?actions": Array.of({
  "name": Identifier,
  "?comment": String
}),
```

Constants

- constant values do not change over time
- used to define values such as
  number of processes
  concrete probability values
  initial number of molecules
- may be left open to be specified externally
- have a certain type (e.g. integer, real, etc.)

```
var ConstantDeclaration = schema({
  "name": Identifier, // unique constant's name
  "type": [ BasicType, BoundedType ], // the constant's type
  "?value": ConstantValue, // the constant's value, of type type
  "?comment": String // optional comment
});
```

Global variables

- values can change over time
- readable/writable by all automata of the model
- value of non-`transient` variables preserved if not changed

```
var VariableDeclaration = schema({
  "name": Identifier, // unique variable's name
  "type": Type, // the variable's type
  "?transient": [ true, false ], // transient variable if present and
      true
  "?initial-value": [ // unrestricted if not present
    null, // the default value of type
    Expression // a constant expression of type type
  ],
  "?comment": String // an optional comment
});
```

Initial values

- restricts initial values of global variables
- compared to `initial-value`, more complex restrictions possible
- in particular, depending on several variables
  e.g. $x = 1 \lor y = 2$

```
"?restrict-initial": {
  "exp": Expression,
  "?comment": String
},
```

Automata specification

- behaviour of JANI files specified by network of *automata*
- these read/write model variables
- contain set of *locations*
- also have their own *local* variables

```
var Automaton = schema({
  "name": Identifier,
  "?variables": Array.of(VariableDeclaration),
  "?restrict-initial": {  "exp": Expression },
  "locations": Locations
  "initial-locations": Array.of(Identifier),
  "edges": Edges
  })
});
```

Locations specification

- automaton always in one given location
- initially, in one of the initial locations
- note: locations are *not* states

```
Locations Array.of({
  "name": Identifier,
  "?invariant": { "exp": Expression },
  "?transient-values": Array.of({
    "ref": LValue,
    "value": Expression,
  }),
}),
```

```
"initial-locations": Array.of(Identifier),
```

Edges specification

- specify changes of modes and variables

```
Edges: Array.of({
  "location": Identifier, // source location
  "?action": Identifier,  // used for synchronisation
  "?rate": {  "exp": Expression  }, // for continuous-tim models
  "?guard": { "exp": Expression }, // when can be executed?
  "destinations": Array.of({ // stochastic choice of locations
    "location": Identifier, // successor locations
    "?probability": { "exp": Expression  }, // branch probability
    "?assignments": Array.of({  // variable assignments of branch
      "ref": LValue, // variable affected
      "value": Expression, // new value
      "?index": Number.step(1),
    }),
  }),
```

Composition specification

- describes how automata interact with each other
- specify *synchronisation vectors*
- favourite synchronisation mechanism of Hubert Garavel
- subsumes CCS, CSP and others
- in `elements`, an automaton can be used multiple times
  a new copy is used each time
- `input-enable`: adds self-loops with given action if needed

```
var Composition = schema({
  "elements": Array.of({
    "automaton": Identifier, // the name of an automaton
    "?input-enable": Array.of(Identifier) // make input enabled
  }),
  "?syncs": Array.of({
    "synchronise": Array.of([ Identifier, null ]),
          // a list of action names or null, same length as elements
    "result": Identifier,
          // an action name, the result of the synchronisation
  }),
});
```

Properties

- properties of the model, e.g. in PCTL/PLTL, usw.
- also contains PRISM-style filters
- do not assume all tools will immediately support all property types specified

```
var Property = schema({
  "name": Identifier, // the unique property's name
  "expression": PropertyExpression // the state-set formula
  "?comment": String // an optional comment
});
var PropertyExpression = schema([
  [\dots],
  { // until / weak until
    "op": [ "U", "W" ],
    "left": schema.self,
    "right": schema.self,
    "?step-bounds": PropertyInterval,
    "?time-bounds": PropertyInterval,
    "?reward-bounds": Array.of({
      "ref": Expression,
      "accumulate": Array.of([ "steps", "time" ]),
      "bounds": PropertyInterval
    })
  }
]);
```

Semantics

- JANI models represent a model according to its `type` specification
- here, we only consider MDPs
- some constructs depend on model type used
  e.g. `rate` must not be used for discrete-time models
  e.g. clocks are only allowed for timed automata variants

Semantics: state space

- consider non-transient *global* variables
- consider automata locations
- consider non-transient *local* variables
  one set for each copy of `automaton` mentioned in the composition
- state: mode of each automata plus value assignment for all variables
  e.g. one automaton with modes m1, m2, local variable boolean z
  global variables $\{x, y\}$, x integer, y boolean
  valid states:
  $(m1, x = 2, y = \mathit{false}, z = \mathit{false})$,
  $(m1, x = 1, y = \mathit{false}, z = \mathit{true})$,
  $(m2, x = -7, y = \mathit{true}, z = \mathit{false})$,
  ...

Initial states

- initial modes possible depending on `initial-modes`
- for variable, potentially use `initial-value` of variable, if given
- complex expressions possible using `restrict-initial`
- initial states: states fulfilling conjunction of restrictions
  e.g. one automaton
  no local variables, global variables x, y
  singleton set of initial nodes $m1$
  `initial-value` of x is 2
  `restrict-initial` states $x = 2 \rightarrow y = \mathit{false}$
  then the only initial state is $(m1, x = 2, y = \mathit{false})$

Labeling function

- labels each state with value of non-transient global variables
- transient global variables visible using `transient-values` of locations
- in properties, use expressions over states
  e.g. $(m1, x = 2, y = \mathit{false}, z = \mathit{false}) \models x > 1 \wedge \neg z$
  $\mathbb{P}_{\max=?}(\mathbf{F} \ \models x > 1 \wedge \neg z)$

Actions

- *Act* derived from `actions` definition plus one "invisible" action

```
"?actions": Array.of({
  "name": Identifier,
  "?comment": String
}),
```

Transitions - single automaton

- consider a given state s
- an edge of a given automaton is *enabled* if `location` agrees and `guard` fulfilled
- for single automaton, each `destination` chosen by given `probability`
- for given `destination`, `assignments` change state variables leading to an according successor state

```
Edges: Array.of({
  "location": Identifier, // source location
  "?action": Identifier,  // used for synchronisation
  "?rate": {  "exp": Expression  }, // for continuous-tim models
  "?guard": { "exp": Expression }, // when can be executed?
  "destinations": Array.of({ // stochastic choice of locations
    "location": Identifier, // successor locations
    "?probability": { "exp": Expression  }, // branch probability
    "?assignments": Array.of({  // variable assignments of branch
      "ref": LValue, // variable affected
      "value": Expression, // new value
      "?index": Number.step(1)
    }),
  })
```

Transitions - synchronisation

- consider state s
- for each `automaton` of `elements` consider all enabled edges
- the `composition` specifies the *synchronisation vector*

```
var Composition = schema({
  "elements": Array.of({ "automaton": Identifier }),
  "?syncs": Array.of({
    "synchronise": Array.of([ Identifier, null ])
    "result": Identifier
  })
});
```

e.g.

Automaton 1	Automaton 2	Automaton 3	Result
a	a	a	a
b!	b?	`null`	τ

- each row states possible synchronisation
- e.g. first row: all three automata perform *a*-labelled edge
- e.g. second row: automaton 1 and 2 perform b! resp. b?; 3 not involved
- nondeterministic choice between sets of edges executed together

Transitions - executing edges

- effects of all non-`null` edges executed together
- probabilities of destinations multiplied
- e.g. assume have $\{e1, e2, e3\}$ with
  $destinations(e1) = [d1 \mapsto 0.5, d2 \mapsto 0.5]$
  $destinations(e2) = [d3 \mapsto 0.3, d4 \mapsto 0.4, d5 \mapsto 0.3]$
  $destinations(e3) = [d6 \mapsto 0.25, d7 \mapsto 0.75]$
- then combined destinations are
  $destinations(e1 : e2 : e3) = [$
  $d1 : d3 : d6 \mapsto 0.5 \cdot 0.3 \cdot 0.25,$
  $d1 : d3 : d7 \mapsto 0.5 \cdot 0.3 \cdot 0.75,$
  $\ldots$
  $]$
- all assignments of destinations executed together
- thus, synchronised edges *must not* write to same variable
- (except if using `index` feature, not discussed here)

EPMC

- mainly written in Java, with some parts written in C
- modular approach to perform model checking
- targets at using proven techniques from software engineering
  e.g. appropriate use of patterns (builder, delegate, etc.)
- divided into core parts and plugins (division not completely fixed)
- uses Maven, Ant, and make for the build process
- Eclipse should be developed for development
- uses external component where appropriate
- free available on GitHub
  `https://github.com/ISCAS-PMC/ePMC`
- open source, released under GPLv3