Mec 5

A tool to manipulate finite relations

LaBRI

This manual documents Mec 5.4.

Contributors to this manual are Aymeric Vincent and Farès Chucri.

Table of Contents

1 Introduction

Mec 5 is a model-checker. Its implementation is based on the well-known data structure called “binary decision diagrams” (BDDs).

Mec 5 is basically a tool which will allow you to compute fixpoints over relations of arbitrary arity. However, it will of course allow you to define relations based on descriptions of models. These models can currently only be given as AltaRica models. Former Mec 4 models can be loaded, synchronized, and displayed but nothing useful can be extracted from them with Mec 5 yet.

2 First session

Upon startup, Mec 5 displays a small welcome message and awaits input from the user.

```
Type :help to get a list of commands.          Mec 5.4

[mec]
```

At the input prompt, the user can either enter a utilitary command which is prefixed by a semi-colon (`:`) or a formula from the specification language.

A formula of the specification language can extend over several lines. In this case, Mec 5 changes the prompt to indicate that it is waiting for more input from the user. Moreover, every definition of an object triggers the printing of its type. For example:

```
[mec] R(x) :=
.                x = true;
        R: (bool) -> bool
[mec] cst := false;
        cst: bool
[mec]
```

If Mec 5 was compiled with support for a `readline`-compatible library, it will allow the user to edit its input and use the history facilities of that library. Please refer to the readline library's documentation for further information.

3 Computing relations

The main goal of Mec 5 is to compute relations over finite domains. In every place where it made sense, we tried to stick to the syntax of the AltaRica language, so as to provide a coherent user experience. However, there are a couple of exceptions, see Section 3.4 [Divergences with the AltaRica language], page 6.

3.1 Domains

Base domains from the AltaRica language are supported and retain the same syntax. They are: `bool`, finite integer arrays like '`[-5, 5]`', enumerations like '`{ foo, bar }`'.

It is also possible to refer to domains via user-defined names thanks to the `domain` primitive.

Moreover, the following domains are added: for every AltaRica node `n` loaded into Mec 5, three domains `n!c`, `n!c`, and `n!ev` are defined, which respectively type *configurations*, *super configurations* and *event vectors* of the node `n`.

3.2 Expressions

Except for the two remarks at the beginning of this section, the syntax of AltaRica expressions is respected to the highest extent in formulas of the specification language; it has been enriched by predicates and first-order quantifiers. We describe here only the extensions specific to the specification language.

Predicates or first-order quantifiers can now be used as any other unary expression.

For first-order quantifiers, the pairs of characters ⟨<>⟩ and ⟨[]⟩ are used to denote respectively existential and universal quantifiers.

Any symbol introduced by such a quantifier can be typed explicitly; it even has to be typed if the subformula doesn't provide enough information for the type to be inferred. For example, '`<x : D> P(x)`' is read $\exists x.(x \in \mathcal{D} \wedge P(x))$.

Using a predicate is done by enclosing its parameters within parentheses and separating them with commas, like a function call in several languages. This is motivated by the fact that a predicate is a function from the types of its arguments to `bool`.

3.3 Defining new objects

In addition to loading AltaRica nodes, the user can define new domains, constants, and relations. Relations are the objects used as predicates.

AltaRica nodes

Loading an AltaRica model '`n`' with `:ar-load` creates two new domains and two new relations:

`n!c`	the type of its configurations
`n!sc`	the type of its super configurations
`n!ev`	the type of its event vectors
`n!t`	its transition relation (subset of `n!c x n!ev x n!c`)

`n!st` its super transition relation (subset of `n!c x n!ev x n!c`)

`n!init` its set of initial states (subset of `n!c`)

Note that the *epsilon* event is represented by an empty string which can be written as a double double-quote ⟨""⟩.

Event vectors can be accessed in a structured manner. An event vector is made of a local event name and one event name per leaf subnode. Each component is accessed with a dotted notation, using the instance names of the subnodes to navigate down to the leaf subnodes. To access the local event name, the notation is to suffix the event vector's name with a dot ⟨.⟩.

Examples of valid expressions are: '`e. = "on"`', '`e.A.S = "failure"`'.

Type definitions

Defining a new domain is done with the same syntax as in AltaRica, except that the trailing semi-colon (⟨;⟩) is mandatory.

```
[mec] domain state = { empty, full };
[mec]
```

Constant definitions

For example, the definition of a new constant is written like this:

```
[mec] cst := 25;
        cst: integer
[mec]
```

Please note that the current version of Mec 5 doesn't allow first-order quantifiers nor predicates in the expression defining the constant. All the other operators available in expressions are supported.

Relation definitions

A relation can be defined by giving an expression that will constrain its parameters:

```
[mec] R(x : [0, 10]) := x = 2 | x = 4;
        R: ([ 0, 10 ]) -> bool
[mec] :display R
(4)
(2)
[mec]
```

It is also possible to define a relation through the use of a least or greatest fixpoint.

Here is an example of a least fixpoint:

```
[mec] S(x : [0, 10]) += x = 4 | S(x + 1);
        S: ([ 0, 10 ]) -> bool
[mec] :display S
(4)
(3)
(2)
(1)
```

```
(0)
[mec]
```

Systems of equations can also be solved, which gives a way to alternate fixpoints. The integer i that can be embedded within operators encodes the parity associated with the relation being defined. Here is a small table relating operators and parities.

operator	**parity**
+i=	$2i + 1$
-i=	$2i$

The biggest parity in a play of the induced game determines the winning condition. In the absence of the integer i, it defaults to 1.

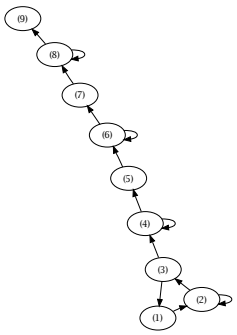
In order to avoid cluttering the global name space and to save memory, it is possible to use the `local` keyword to make a relation local to the given system of equations.

Fixpoint equations are ordered depending on their parities; the order in which they are given doesn't matter.

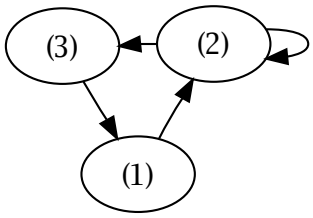
Here is an example that comes from the Toupie tool:

```
[mec] domain vertex = [1, 9];
[mec] g(S : vertex, T : vertex) += (T = S + 1) |
.               (T = S & (S = 2 | S = 4 | S = 6 | S = 8)) |
.               (T = 1 & S = 3);
        g: ([ 1, 9 ], [ 1, 9 ]) -> bool
[mec] odd(S : vertex) += <n : [0, 4]>(S = n + n + 1);
        odd: ([ 1, 9 ]) -> bool
[mec] begin
. local aux(V : vertex) +1= <W : vertex>(g(V, W) & aux(W)) |
.                           <W : vertex>(g(V, W) & odd(W) & tau(W));
.       tau(U : vertex) -2= aux(U);
. end
        tau: ([ 1, 9 ]) -> bool
[mec] :display tau
(3)
(2)
(1)
[mec] gtau(S : vertex, T : vertex) := g(S, T) & tau(S) & tau(T);
        gtau: ([ 1, 9 ], [ 1, 9 ]) -> bool
[mec]
```

In this example, `g` represents the transition relation of a graph which is basically a line with self loops at every even vertex, and a strongly connected component.



The equation system looks for the strongly connected component which contains at least an odd vertex; it is represented by the set of vertices `tau`, and the `gtau` transition relation is the restriction of `g` to the set of vertices `tau`.



It is easy to obtain the graphical representation of these graphs with the GraphViz package and with the help of the `:graph-print-dot` command. See [graph-print-dot], page 8.

3.4 Divergences with the AltaRica language

The specification language of Mec 5 is expected to offer for its expressions a superset of the AltaRica expressions. The major differences are listed here.

Operators

Logical operators are not available in their english form, so as to limitate namespace pollution. For example, '`x or y`' is accepted in an AltaRica model but not within a Mec 5 specification; it should be written '`x | y`'.

Identifiers

In order to allow primed variables, the quote character ⟨'⟩ can be part of a Mec 5 identifier.

An identifier starts with a letter or an underscore and may contain letters, underscores, the quote character, and the carret ⟨^⟩ character.

Double-quotes ⟨"⟩ should be used to enclose identifiers which contain any other character. However, even these identifiers shall not contain dots ⟨.⟩ nor double-quotes ⟨"⟩.

4 Commands

You can use the `:help` command to see a list of the available commands. This list of commands may vary depending on your precise version of the tool, or on the compilation options.

Each command may be abbreviated to any of its non-ambiguous prefixes. Because such non-ambiguous prefixes depend on the other commands available in Mec, the use of an incomplete command name is strongly discouraged in scripts. Its intended use is for interactive typing.

4.1 Commands involving defined relations

:display

displays a previously defined constant.

```
[mec] cst := false;
        cst: bool
[mec] :display cst
false
[mec]
```

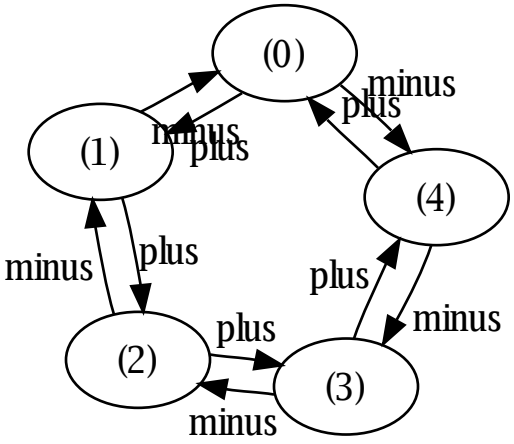
:graph-print-dot

outputs in GraphViz format the graph corresponding to the transition relation given as a parameter. The relation is split into two or three components like this: consider the sequence of types forming the type of the relation. The largest prefix which is also a suffix of this sequence constitute together the components of the relation which represent the nodes of the graph. If there are unused components in between, they will constitute the events and be displayed as such.

```
[mec] R(s : [0, 4], e : {plus, minus}, t : [0, 4]) :=
.         (e = plus) & ((s + 1 = t) | (s = 4) & (t = 0)) |
.         (e = minus) & ((s - 1 = t) | (s = 0) & (t = 4));
        R: ([ 0, 4 ], { plus, minus }, [ 0, 4 ]) -> bool
[mec] :graph-print-dot R > R.dot
[mec]
```

And you could get a PDF file out of the '`.dot`' file with a shell command like:

```
$ neato -Tpdf < R.dot > R.pdf
```



:rel-cardinal

displays the number of elements in the relation given as a parameter.

```
[mec] R(x : [0, 4], y : [0, 4]) := x + y = 4;
        R: ([ 0, 4 ], [ 0, 4 ]) -> bool
[mec] :display R
(0, 4)
(4, 0)
(2, 2)
(1, 3)
(3, 1)
[mec] :rel-cardinal R
cardinal of R: 5
[mec]
```

:rel-size

displays an estimation of the size taken by a given relation. The current output unit is the number of BDD nodes used to represent the relation.

```
[mec] :rel-size R
BDD size of R: 14
[mec]
```

:remove-constant

removes the given constant from the set of defined constants. Useful to free the memory associated with a big relation.

The `:remove-constant` command cannot be abbreviated to avoid the unfortunate loss of an important constant.

4.2 AltaRica commands

:ar-load

loads the given AltaRica file and defines the two domains and relations describing each node in the file.

```
[mec] :ar-load try.alt
[mec]
```

:ar-poset-print-dot

prints the partial order describing the priority relation of the given node's events in GraphViz format.

:ar-tree-print-dot

outputs in GraphViz format a directed acyclic graph representing the hierarchy including and below the AltaRica node given as a parameter. Each node in the graph is an AltaRica node, and two nodes are linked with an arc if and only if the former is a direct parent of the latter. Arcs are labelled by the name of the instance of the subnode in the parent node.

:ar-display

displays some information about the given AltaRica node. The output is very terse and can not be considered yet as a way to save back a node to disk.

```
[mec] :ar-load ../../examples/Peterson+N2.alt
[mec] :ar-display main
node main
        flow turn1 : [ 1, 2 ];
        state level1 : [ 1, 2 ];
sub variables
        flow myLevel : [ 0, 1 ];
        flow left1Level : [ 0, 1 ];
        flow ident : [ 1, 2 ];
        flow turn1 : [ 1, 2 ];
        state co : { A, B, C, D };
        state currentLevel : [ 0, 1 ];
        flow myLevel : [ 0, 1 ];
        flow left1Level : [ 0, 1 ];
        flow ident : [ 1, 2 ];
        flow turn1 : [ 1, 2 ];
        state co : { A, B, C, D };
        state currentLevel : [ 0, 1 ];
relation for event level1To1 has size 58
relation for event  has size 225
relation for event level1To2 has size 58
edon
```

```
[mec]
```

4.3 CEGAR commands

:ar-cegar-init

The syntax of the `ar-cegar-init` command is:

`ar-cegar-init` <*Node*> <*Init*> <*Err*> *[P1 P2 ...]*

- *Node*: An AltaRica Node
- *Init*: A set of initial states
- *Err*: A set of bad states
- *P1 P2 ...*: A list of sets of states

The `ar-cegar-init` command computes an abstraction of *Node* with *Init* as its initial state, *Err* as its final state, and uses *P1 P2 ...* to compute its abstract states.

:ar-cegar-verify

The syntax of the `ar-cegar-verify` command is:

`ar-cegar-verify` <*Node*> *[N]*

- *Node*: An AltaRica Node
- *N*: An integer

The `ar-cegar-verify` command performs a reachability check on the abstraction of node *Node* by running the CEGAR loop until termination if *N* is not specified, otherwise the CEGAR loop will be performed *N* times.

:ar-cegar-print

The syntax of the `ar-cegar-print` command is:

`ar-cegar-print` <*Node*>

- *Node*: An AltaRica Node

The `ar-cegar-print` command print the abstraction of the node *Node* in dot format.

:ar-cegar-statistics

The syntax of the `ar-cegar-print` command is:

`ar-cegar-statistics` <*Node*>

- *Node*: An AltaRica Node

The `ar-cegar-statistics` command print statitics on the abstraction of the node *Node*.

4.4 General purpose commands

:help

displays the list of available commands. It can also take any number of command names as parameters and will then display for each of them its synopsis of and a short message describing it.

```
[mec] :help display
display string
        display the given object
[mec]
```

:set

The `:set` command allows to define options which affect the behaviour of Mec 5. It takes two parameters: the name of the option, and the value to assign to it.

The description of available options can be seen in Section 5.2 [Options], page 14.

Boolean options can be assigned the value `on` or `off`. They can also be set by using only the name of the option as unique parameter to `:set` and unset by doing the same but prefixing the name of the option with `no-`.

```
[mec] :set no-verbose-ev
[mec]
```

Options and their associated value can be displayed with '`:list options`'.

:list

displays a list of formerly defined entities. Without parameter, it displays the set of lists that can be displayed.

```
[mec] :list
Available lists:
ar-nodes        AltaRica nodes
constants       Constants
domains         Domains
options         Options
relations       Memory used by relations
timers          Timers
trans-sys       Transitions systems
cegar-graphs    CEGAR abstractions
[mec]
```

:timers

takes as a parameter one of the three keywords `enable`, `disable`, or `reset`, and allows to evaluate the time taken by various activities in Mec 5. The timers and their associated value are listed with '`:list timers`'. By default, timers are disabled.

:spec-load

loads the given file as a specification file. This file can contain commands or specification formulae, and is interpreted as if its contents were typed interactively, except that some of the informational output is not displayed in this case. E.g. the type of newly defined relations is not displayed.

```
[mec] :spec-load try.spec
[mec]
```

:quit

quits Mec 5.

5 Environment

Mec 5 can help improve the user experience by providing history saving and reloading across sessions, and several options to tune the behaviour of Mec 5.

5.1 The '.mec/' directory

If you create a '`.mec`' directory in your home directory, Mec 5 will start doing two things:

- a file called '`last-session.html`' will be saved in this directory every time you finish a Mec 5 session. It is useful if you forgot what exactly you did and offers highlighting which helps locate user input and so on.

  It currently suffers from a bug where the file will be broken if two sessions of Mec 5 run simultaneously.
- if Mec 5 was compiled with a `readline`-compatible library, it will save the history of sessions in a '`history`' file in this directory, and make this history available through `readline` for future sessions. The '`history`' file is not intended to be read by humans.

5.2 Options

In order to tune the behaviour of Mec 5, some options are available to the user. They can be set with the command `:set`, see [set], page 12. They can be listed with the command '`:list options`', see [list], page 12.

verbose-ev

This boolean option is *on* by default.

When this option is disabled, event vectors of AltaRica models (elements of type `n!ev`) are displayed in a shorter form.

bdd-terms-cache-size

This integer option has a default value of 1 million.

The integer represents the number of entries in the BDD terms' cache. Raising this value will make Mec 5 use more memory but could also improve the performance of Mec 5. See Section 6.2 [BDD terms' cache size], page 18.

mec4-extensions

This boolean options if *off* by default.

If enabled, it will make the underscore character _ a reserved keyword in synchronization tables, as can be loaded by `mec4-load`. This character represents a wild card and allows to synchronize with any event.

debug-mask

This integer option has a default value of 0.

This option makes sense only if Mec 5 was compiled with the `DEBUG` flag, which is not the case by default. Setting this option when Mec 5 is not compiled with the `DEBUG` flag leads to a warning message informing the user that the option has no effect.

Useful integers are the bitwise or of the `DEBUG_` values found in the '`util/debug.h`' file in Mec 5's sources.

To compile a `DEBUG` version of Mec 5, see Section 7.5 [Compiling a DEBUG version], page 21.

cegar-abstraction-type

This option has a default value of *Existential*.

This option defines the type of abstraction computed by [ar-cegar-init], page 11.

Value	Description
Existential	Compute a Existential Abstraction.
Modal	Compute a Modal Abstraction.

cegar-use-concrete-events

This boolean options if *off* by default.

If enabled, it will make the [ar-cegar-init], page 11 command use all the events of the AltaRica node abstracted during the abstract transition relation computation. Otherwise, all events will be abstracted by a single event denoted by " * ".

This option has no effect if the option [cegar-abstraction-type], page 15 is set to *modal*.

cegar-use-transitive-marking

This boolean options if *off* by default.

If enabled, this option will make the [ar-cegar-verify], page 11 command to statistically add modal states to the abstraction. The modal states added depend on the refinement method used.

This option has no effect if the option [cegar-abstraction-type], page 15 is set to *Existential*.

cegar-use-guards-as-predicates

This boolean options if *off* by default.

If enabled, this option will make the [ar-cegar-init], page 11 command to ignore any predicate *P1 P2 ...* passed to it. Instead, the AltaRica transition guards will be used as predicates.

cegar-search-algorithm

This options has a default value of *BFS*.

This option defines the abstraction search algorithm used by [ar-cegar-verify], page 11 to find an abstract counter-example.

Value	Description
BFS	Use the breadth first search algorithm.
DFS	Use the breadth depth search algorithm.
Back_DFS	Use the backward breadth first search algorithm.
Back_BFS	Use the backward depth first search algorithm.

cegar-ce-analysis

This options has a default value of *Forward*.

This option defines the way the counter-example analysis method used by [ar-cegar-verify], page 11 searches for a failure state.

Value	Description
Forward	Search for a failure state from *Init* to *Err*.
Backward	Search for a failure state from *Err* to *Init*.

cegar-use-frontier-co

This boolean options if *off* by default.

If enabled, this option will make the [ar-cegar-verify], page 11 command to search a counter-example from a negative modal state to a positive modal state.

This option has no effect if the option [cegar-abstraction-type], page 15 is set to *Existential*.

cegar-use-super-transition

This boolean option is *off* by default.

When this option is enabled, the [ar-cegar-init], page 11 command will use the super transition relation of AltaRica node to compute the abstract transition relation (instead of its asserted transition relation).

cegar-abstraction-print

This boolean option is *off* by default.

When this option is enabled, the [ar-cegar-verify], page 11 command will print the abstraction as in the dot format at each iteration of the CEGAR loop.

cegar-use-unsafe-sub-ce

This boolean option is *off* by default.

When this option is enabled, the [ar-cegar-verify], page 11 command will analyse the first shortest unsafe part of the counter-example taken into consideration at each iteration.

This option has no effect if the option [cegar-abstraction-type], page 15 is set to *Existential*.

cegar-use-unasserted-nodes

This boolean option is *off* by default.

When this option is enabled, the [ar-cegar-init], page 11 command will not constraint the abstract nodes computed with the AltaRica node assertion.

cegar-abstraction-refinement-algorithm

This option has a default value of *Post_Sigma*.

This option defines the method used by [ar-cegar-verify], page 11 to refine the set of abstract states of an abstraction when a spurious abstract state is detected.

Value	**Description**
Post_Sigma	Use the *Post_Sigma* Refinement heuristic.
Direct	Use the *Direct* Refinement heuristic.
Pre_Sigma	Use the *Pre_Sigma* Refinement heuristic.
Back_Direct	Use the *Back_Direct* Refinement heuristic.
CSG	Use the *CSG* Refinement heuristic.
Post_Pre	Use the *Post_Pre* Refinement heuristic.

6 Performance

There are a few ways in which you can improve the performance of Mec 5.

6.1 Code optimisation

By default, Mec 5 is compiled with conservative compiler settings which help produce useful bug reports from users.

With gcc, you can win as much as about 20% speed by changing the flags passed to the compiler when compiling Mec 5 by replacing the default `-g -O` option with `-O3` for example. See Section 7.4 [Modifying the compiler flags], page 21.

Check the documentation of your compiler to see which flags produce faster code.

6.2 BDD terms' cache size

The '`bdd-terms-cache-size`' option allows to set the size of the BDD terms cache. This cache keeps the last computations performed on BDDs, and it is the main reason for the speed of computations on BDDs. Growing this cache can vastly improve the performance of computations.

The unit of this option is a number of entries in the cache, and a typical value is currently in the order of a million. See [bdd-terms-cache-size], page 14.

6.3 Good practice

6.3.1 Removing unused relations

The `:remove-constant` command allows to dispose of a previously defined constant. It can free at most as many nodes as there are in this relation. Possibly fewer because relations share nodes among themselves.

If you are computing several relations in sequence, and you don't need some of the intermediary relations, you can re-use the same relation name, and it will automatically dispose of the previous value for this relation.

6.3.2 Using quantifiers as late as possible

In a formula, for various reasons related to the way formulae are evaluated, it is more efficient to introduce local variables with quantifiers the latest possible. For example, the accessible state space can be equivalently computed by the two following fixpoints, but with the current implementation, the second version will be computed faster:

```
[mec] R(s) += N!init(s) | <s'><e>(R(s') & N!t(s', e, s));
```

compared with

```
[mec] R(s) += N!init(s) | <s'>(R(s') & <e>N!t(s', e, s));
```

6.4 AltaRica-specific performance improvement

Because Mec 5 is mostly used to study AltaRica models, the traditional handling of these models has been extended to make it possible for the users to perform faster computations.

For any AltaRica node '`n`', we saw that two relations are provided which permit a full exploration of the AltaRica model:

`n!init` the initial configurations of the node '`n`'

`n!t` the transition relation of the node '`n`'

We shall see in this section that actually a few more relations are provided which help the user avoid the often huge and hardly tractable '`n!t`' transition relation.

6.4.1 Subevents' transition relations

If a node has several subevents, it is possible to manipulate separately the transition relation of each subevent. The syntax for accessing the transition relation of the subevent '`e`' of node '`n`' is '`n.e!t`'. This relation has the same type as '`n!t`'.

For example, in order to compute the successors of any initial configuration by a '`failure`' transition, we can use the traditional approach like this:

```
[mec] S(s) := <s'>(A!init(s') & <e>((e. = failure) & A!t(s', e, s)));
```

... but we can also use the more efficient approach which avoids entirely the complexity of the transition relation for the other events:

```
[mec] S(s) := <s'>(A!init(s') & <e>A.failure!t(s', e, s));
```

The same mechanism works for the "super transition relation" defined below.

6.4.2 Super transition relation

For advanced users only! Semantics may be affected.

Often, the transitions induce a very small and simple transition relation representing for example the incrementation of a counter, or the flipping of a boolean variable. However, for the AltaRica semantics to be respected, any transition relation manipulated by Mec 5 has to respect the constraints imposed by the assertion of its node and subnodes.

After constraining the transition relation with the assertion, the size of its representation by a BDD often becomes huge, to the point of not being manageable.

The only case in which it really matters to apply the assertion's constraint at every step of the computation of the semantics of the model is if the model contains priorities over events.

If your model uses event priorities, do not use the feature described in this subsection!

When Mec 5 computes a relation, it makes sure that the relation is constrained by the set of values allowed by its type: if a relation is defined over the set '`[1, 3]`', there is no way this relation will contain anything outside '`[1, 3]`'. The same goes for AltaRica configurations, and the constraint of the type '`n!c`' is the assertion relation of the AltaRica node '`n`'.

This means that as long as you use the transition relation in a monotonic way, you are safe to apply the assertion's constraint at the end of the computation.

Reachability can be computed with this over approximation of the transition relation because at each step of the fixpoint computation, the type constraint (the assertion) is applied to the current set of reachable configurations. This ensures that paths that are forbidden by the assertion relation cannot be taken to reach an unreachable configuration which itself would be allowed by the assertion relation.

This overapproximation of the transition relation is accessible through the '`n!st`' relation for an AltaRica node '`n`'. It has the same type as '`n!t`' and also offers the subevents' transition mechanism described in the previous subsection.

7 Compiling Mec 5

The process of compiling Mec 5 is simplified by the use of a script. Here is how you can proceed to get a working version of mec from its sources.

We will note the shell prompt with the dollar ⟨$⟩ character.

7.1 Prerequisites

Mec 5 needs a C compiler, the `flex` lexical analyzer generator, and a decent `yacc` or `bison` parser generator.

Mec 5 will offer improved functionality if a `readline`-compatible library with its necessary header files is installed on the system.

Mec 5 has been tested to compile out of the box as long as these prerequisites are fulfilled on Sun Solaris, NetBSD, OpenBSD, Mac OS X, various Linux distributions, and Cygwin under Microsoft Windows.

7.2 Default compilation

The first step is to untar the sources with a command like:

```
$ gzip -cd mec-5.4.tar.gz | tar xf -
```

This should create a directory called '`mec-5.4`'. The compilation script must be launched from the '`mec-5.4/build/`' directory:

```
$ cd mec-5.4/build
$ ./do-build
```

Then, when compilation is finished, the `mec` executable can be found in a subdirectory inside '`mec-5.4/build/`', whose name depends on the machine you compiled Mec 5 on. You should be able to copy this executable wherever you wish. For example:

```
$ cd mec-5.4/build
$ ./do-build
Checking where distribution is... found in ../..
Checking whether make is GNU make... no
[...]
$ ls
darwin-powerpc  do-build        netbsd-powerpc
$ cd netbsd-powerpc
$ ./mec

Type :help to get a list of commands.          Mec 5.4

[mec]
```

7.3 The compilation process

The '`do-build`' script is very handy to get quickly a `mec` executable that works. However, if you use Mec 5 from its CVS sources for example, you might want to know what simple steps this script performs.

1. Prepare a directory under the '`build`' directory and move to this directory. Nothing outside this directory will be touched.
2. Always from this directory, run the `configure` script which is located at the root of the Mec 5 sources. This creates a '`Makefile`' or '`GNUmakefile`' suited to your `make` tool.
3. You may want to run '`make clean`' to rebuild all the object files from scratch. '`make clean`' will remove only files that should be regenerable automatically. It should not touch any file you put by hand in this directory.
4. You should compute the dependencies between files with '`make depend`' (this step is useless with GNU make).
5. You can now type '`make`' to produce the `mec` executable.

The whole point of knowing this (classical) procedure is that you can avoid many of the steps when for example only one source file was modified and you want to test the change.

7.4 Modifying the compiler flags

The compiler flags are stored in the same directory where `mec` is compiled. They are either in the file '`GNUmakefile`' or '`Makefile`', depending on whether your default `make` utility is GNU Make or not.

1. Edit this file and locate the line starting with '`CFLAGS=`' near the top of the file.
2. Replace the default by the options you wish
3. Issue a '`make clean`' to ensure every source file is recompiled with the new flags
4. Issue a '`make`' to get a new `mec` executable compiled with your flags

7.5 Compiling a `DEBUG` version

You may want to get verbose output about what is going on inside Mec 5. For this you need to compile a special version of Mec 5, and change the value of the `debug-mask` option. See [debug-mask], page 14.

For this, in the Makefile, you need to uncomment the line

```
#CPPFLAGS+=      -DDEBUG
```

which means to remove the sharp sign at the beginning:

```
CPPFLAGS+=       -DDEBUG
```

You can follow the indications of the previous section to locate the Makefile. See Section 7.4 [Modifying the compiler flags], page 21.

Don't forget to '`make clean`' so that every file will be compiled in `DEBUG` mode.

Full Index