χChek RC2 User Manual

Documentation maintained by
Jocelyn Simmonds and Arie Gurfinkel

xchek@cs.toronto.edu
Department of Computer Science
40 St George Street
University of Toronto
Toronto, Ontario, Canada
M5S 2E4

Contents

Chapter 1

Introduction

χChek is a multi-valued symbolic model checker [CDG02]. It is a generalization of an existing symbolic model checking algorithm to an algorithm for a multi-valued extension of CTL (χCTL). Given a system and a χCTL property, χChek returns the *degree* to which the system satisfies the property. By multi-valued logic we mean a logic whose values form a *finite quasi-boolean distributive lattice*. The meet and join operations of the lattice are interpreted as the logical *and* and *or*, respectively. The negation is given by a lattice dual-automorphism with period 2, ensuring the preservation of involution of negation ($\neg\neg a = a$) and De Morgan laws.

Multi-valued model checking generalizes classical model checking and is useful for analyzing models where there is uncertainty (e.g. missing information) or inconsistency (e.g. disagreement between different views). Multi-valued logics support the explicit modeling of uncertainty and disagreement by providing additional truth values in the logic. χChek works for any member of a large class of multi-valued logics. It's model of computations is based on a generalization of Kripke structures, where both atomic propositions and transitions between states may take any of the truth values of a given multi-valued logic. Properties are expressed in χCTL, a multi-valued extension of the temporal logic CTL.

Additional information about the algorithms and the data structures used by χChek is available in [Gur02, CDEG03, CGD$^+$06]

The main features of χChek are the following:

- Multi-valued logics: 2-valued, 3-valued, upset, 4-valued disagreements, etc. Users can define their own logics.
- Multiple model formats: reads SMV and GCLang models. Multi-valued models are specified in XML.
- Proof-like counterexamples: proof rules are used to generate counterexamples. Counterexamples can be generated statically or dynamically, and various viewers are available.

This document is structured as follows:

- Chapter 2: Installation
- Chapter 3: Input (models, algebras and properties)
- Chapter 4: Guide to the χChek User Interface (loading models, CTL history files, counterexample viewers and preferences)
- Chapter 5: Tutorial (model checking, vacuity detection and query checking)

Known bugs and limitations:

- χChek sometimes runs out of memory when statically computing a counterexample.

- χChek is a "best-effort" implementation. We recommend restarting χChek before loading a new model.

Collaborators:

Marsha Chechik, Benet Devereux, Steve Easterbrook, Arie Gurfinkel, Kelvin Ku, Shiva Nejati, Viktor Petrovykh, Jocelyn Simmonds, Rohit Talati, Anya Tafliovich, Christopher Thompson-Walsh, Kapil Shukla, and Xin (John) Ma.

Chapter 2

Installation

Prerequisites

- Linux-based OS (Fedora Core 4-7, and Ubuntu 7 are known to work).
- Java SE 6.
- Apache ant version $\geq$ 1.6.5 (only for compilation)

Installation

1. Download χChek distribution archive.
2. Extract the χChek archive to create a directory `xchek`, with the following structure:

   build.xml χChek ant buildfile.

   bin scripts for running χChek.

   doc documentation. Includes JavaDoc API and this doucment

   jsrc the source code

   lib bindings to C libraries

   ext 3rd-party Java libraries

   examples example models and properties
3. (Optional) Recompile by running `ant`
4. Run `bin/xc` to execute χChek.

Notes

The amount of memory given to χChek is controlled by setting environment variables `MEMORY_MIN` and `MEMORY_MAX`. For example, in bash,

```
# MEMORY_MIN=512m MEMORY_MAX=1024m ./bin/xc
```

will start χChek with 512MB of available RAM, and will allow the process to use up to 1024MB.

Chapter 3

Input

This chapter describes the syntax of model-specification languages, the syntax of algebra (multi-valued logic) specifications, and the syntax of temporal logic properties.

χChek supports models specified in either a simplified version of the SMV [CCG+02] modeling language, or the Guarded Command Language (GCLang). Arbitrary multi-valued models are specified in XML, by explicitly describing the model's Kripke structure (i.e., states and transitions). These XML files also include a reference to the algebra used to describe the model.

χChek is distributed with some of the more commonly used multi-valued logics, like Boolean — the classical 2-valued logic, Kleene — used for vacuity checking [GC04], 4-valued disagreements logic — used for reasoning about model disagreements [EC01], upset — used for query checking [GCD03], etc.

3.1 Models

SMV and GCLang models are easier to read and maintain, since high-level instructions are used to specify the model. XML models are easier to generate in an automated fashion. Also, this is the only way to explore the full generality of multi-valued model checking in χChek.

3.1.1 SMV

Compiler: SMV Model Compiler (Flat)
`edu.toronto.cs.smv.parser.SmvCompiler`

SMV is the specification language of NuSMV [CCG+02]. Use the following steps to generate an input model for χChek:

1. Specify your model using SMV. This language is described in the NuSMV User Manual, which can be found on the NuSMV website (`http://nusmv.irst.itc.it/`).
2. Once satisfied with the model, flatten it using NuSMV. Modules and processes are instantiated when a SMV model is flattened. Command:

   ```
   %> NuSMV -ofm flat_smv_file.smv normal_smv_file.smv
   ```

3.1.2 GCLang

Compiler: GCLang Compiler
`edu.toronto.cs.gclang.parser.GCLangCompiler`

GCLang is a simple guarded command language. In this type of language, statements have guards. The guard is a proposition, which must be true before the statement is executed. If the guard is false, the statement will not be executed.

GCLang models are divided into four parts:

- NAME: model name
- VAR: variable declaration block
- INIT: variable initialization block
- RULES: guarded commands that specify the behavior of the model

The GCLang syntax accepted by χChek is the following (grouped by part):

NAME

```
start ::                                                -- main model structure
    'NAME' varname varBlock initBlock rulesBlock        -- varname is the model's name
```

VAR

```
varBlock ::                                             -- variable declaration block
    'VAR' ( varDecl )+

varDecl ::
    varname ':' type ';'                                -- e.g., p10 : boolean;

type ::
    'boolean'
    | textSet

textSet ::                                              -- e.g., {0, 1, 2, 3}
    '{' textOrNumberValued ( ',' textOrNumberValued )* '}'

textOrNumberValued ::
    varname
    | number

varname :: atom(atom)*

number :: digit(digit)*

atom :: ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'\\'|'$'| '#'| '-')*

digit :: '0'..'9'
```

INIT

```
initBlock ::                                            -- variable initialization
    'INIT' expr ( ';' | )

expr ::
    implExpr                                            -- initialize using an expression
    | setExpr                                           -- initialize using a set

setExpr ::
    { setElement ( ',' setElement )* }

setElement ::                                           -- sets contain text or numbers
    textOrNumber
```

```
textOrNumber ::
    varname
    | number

implExpr ::
    basicExpr
    | '(' implExpr ')'
    | implExpr '|' implExpr                          -- disjunction
    | implExpr '^' implExpr                          -- exclusive or
    | implExpr '&' implExpr                          -- conjunction
    | '!' implExpr                                   -- negation
    | implExpr '->' implExpr                         -- implication
    | implExpr '<->' implExpr                        -- if and only if
    | implExpr '=' implExpr                          -- equal
    | implExpr '!=' implExpr                         -- not equal
    | implExpr '<' implExpr                          -- less than
    | implExpr '>' implExpr                          -- greater than
    | implExpr '<=' implExpr                         -- less or equal than
    | implExpr '>=' implExpr                         -- greater or equal than
    | implExpr '%' implExpr                          -- module
    | implExpr '+' implExpr                          -- addition
    | implExpr '-' implExpr                          -- subtraction
    | implExpr '*' implExpr                          -- multiplication
    | implExpr '/' implExpr                          -- division

basicExpr ::
    varname
    | number
    | boolConstant
    | ( expr )

boolConstant ::
    'true'
    | 'false'
```

RULES

```
rulesBlock ::
    'RULES' ( guardedCommand )+                      -- rules that specify model behavior

guardedCommand ::
    guard ':' command

guard :: expr

command ::
    sequenceCommand

sequenceCommand ::                                   -- statement concatenation
    choiceCommand ( ';' choiceCommand )*

choiceCommand ::
    atomicCommand ( '||' atomicCommand )*            -- non-deterministic choice

atomicCommand ::
    assign                                           -- assignment
    | 'SKIP'                                         -- empty instruction, do nothing
    | ite                                            -- if-then-else
    | '(' command ')'                                -- group of commands

assign ::
    varname ':=' expr
```

```
ite ::
    'if (' expr ')' ( 'then' | ) iteBody                    -- 'then' is optional

iteBody ::
    command 'else' command 'fi'
```

Note: Usual precedence rules apply and comments start with "`--`".

EXAMPLE: the Peterson mutual exclusion algorithm:

```
1  NAME peterson
2  VAR
3    s    : boolean;
4    y1   : boolean;
5    y2   : boolean;
6    pc1  : {0,1,2,3};
7    pc2  : {0,1,2,3};
8  INIT
9    !s & !y1 & !y2 & pc1=1 & pc2=1;
10 RULES
11 -- guards for process 1
12   pc1=1 :
13         y1 := true; s := true; pc1 := 2
14   pc1=2 :
15         if (!y2 | !s)
16           pc1 := 3
17         else
18           skip
19         fi
20   pc1=3 :
21         y1 := false; pc1 := 1
22 -- guards for process 2
23   pc2=1:
24         y2 := true; s := false; pc2 := 2
25   pc2=2 & ((!y1) | s) :
26         pc2:=3
27   pc2=3:
28         y2 := false; pc2 := 1
```

In this example, the model's name is `peterson` (line 1). Five variables are defined: `s, y1, y2, pc1` and `pc2` (lines 3 – 7). The variable `s` is used to keep track of which process should enter the critical section. Variable `y1` (`y2`) is used by the first (second) process to indicate that it is in the critical section. Variable `pc1` (`pc2`) is used to store the state of the first (second) process. These variables are defined by enumeratation: `1` = not in critical section, `2` = trying to get into critical section, `3` = in critical section. The value `0` is included to make the booleanization of `pc1, pc2` easier. Line 9 initializes `s, y1` and `y2` to false, and `pc1, pc2` to `1`. Finally, the mutual exclusion algorithm is described using guarded commands (lines 10 – 28).

3.1.3 XML

Compiler: XMLXKripkeModelCompiler `edu.toronto.cs.xkripke.XMLXKripkeModelCompiler`

Multi-valued models are specified in XML, by mapping the model to a GXL graph that represents the model's behavior. Nodes are labeled by values of the atomic propositions, and the edges are labeled by logic values. The algebra used to encode the model is also specified explicitly. In this section, we describe how a model is encoded. See Section 3.2 for algebra encoding. This input format has been designed for automated generation, not direct user manipulation.

A model encoding is divided into four parts:

- Namespace declaration.
- Link to logic used to encode the model.
- Nodes: one `<node>` tag per model state. Nodes must have unique identifiers within the model. A node can be defined as initial by adding the following attribute to the `<node>` tag: `xbel:initial='true'`. The `<node>` tag has nested `<attr>` tags - one for each model variable. These `<attr>` tags define the state.
- Edges: one `<edge>` tag per model transition. The nested `<attr>` tag defines the logic value of the transition.

Basic structure of a model file:

```
<gxl xmlns:xbel='www.cs.toronto.edu/xbel' xmlns:xlink='xlink'>
 <graph ID='user_defined_id' edgemode='directed'>

   <!-- LINK TO LOGIC -->
   <xbel:logic xlink:type='simple' xlink:href='link_to_logic_used_to_encode_model'/>

   <!-- MODEL -->

     <!-- NODES -->
     <!-- one node tag per model state -->

     <!-- interpreted as: node_id = model_variable1 = logic_value &
          model_variable2 = logic_value & ... & model_variable_n = logic_value -->
     <node ID='node_id1'>
       <!-- one attr tag per model variable -->

       <attr type='prop' name='model_variable1' value='logic_value' />
       <attr type='prop' name='model_variable2' value='logic_value' />
       <attr type='prop' name='model_variable3' value='logic_value' />
       ...
       <attr type='prop' name='model_variable_n' value='logic_value' />
     </node>

     <!-- an initial state -->
     <node ID='node_id1' xbel:initial='true'>
       <!-- one attr tag per model variable -->

       <attr type='prop' name='model_variable1' value='logic_value' />
       <attr type='prop' name='model_variable2' value='logic_value' />
       <attr type='prop' name='model_variable3' value='logic_value' />
       ...
       <attr type='prop' name='model_variable_n' value='logic_value' />
     </node>


     <!-- EDGES -->
     <!-- one edge tag per model transition -->
```

```
    <!-- interpreted as: there is a 'logic_value' transition from 'node_id1' to 'node_id2' -->
    <edge from='node_id1' to='node_id2'>
      <!-- indicates the logic value of the transition -->
      <attr name='weight' value='logic_value'/>
    </edge>

  </graph>
</gxl>
```

EXAMPLE:

Now we will show how to encode the model shown in Fig. 3.1. This model has been encoded in the 4-valued disagreements logic shown in Fig. 3.2.

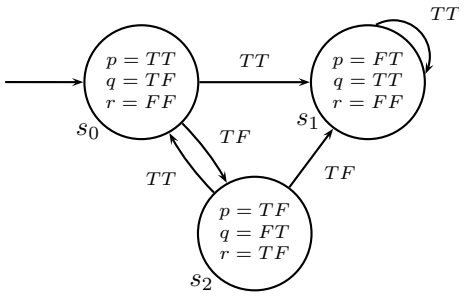

Figure 3.1: A model encoded in the 4-valued disagreements logic.

This is the encoding for this model:

```
<gxl xmlns:xbel='www.cs.toronto.edu/xbel' xmlns:xlink='xlink'>
 <graph ID='model_example' edgemode='directed'>

   <!-- LINK TO LOGIC -->
   <xbel:logic xlink:type='simple' xlink:href='examples/xml/4val-logic.xml'/>

   <!-- MODEL -->
     <!-- NODES -->
     <node ID='s0' xbel:initial='true'>
       <attr type='prop' name='p' value='TT' />
       <attr type='prop' name='q' value='TF' />
       <attr type='prop' name='r' value='FF' />
     </node>
     <node ID='s1'>
       <attr type='prop' name='p' value='FT' />
       <attr type='prop' name='q' value='TT' />
       <attr type='prop' name='r' value='FF' />
     </node>
     <node ID='s2'>
       <attr type='prop' name='p' value='TF' />
       <attr type='prop' name='q' value='FT' />
       <attr type='prop' name='r' value='TF' />
     </node>

     <!-- EDGES -->
     <edge from='s0' to='s1'>
       <attr name='weight' value='TT'/>
     </edge>
     <edge from='s0' to='s2'>
```

```
27          <attr name='weight' value='TF'/>
28        </edge>
29        <edge from='s1' to='s1'>
30          <attr name='weight' value='TT'/>
31        </edge>
32        <edge from='s2' to='s0'>
33          <attr name='weight' value='TT'/>
34        </edge>
35        <edge from='s2' to='s1'>
36          <attr name='weight' value='TF'/>
37        </edge>

38    </graph>
39  </gxl>
```

An explanation of the various items in the model encoding file:

1-2: GXL namespace declaration, the graph is directed and its id is "model_example".

4: location of the logic used to encode the model.

6–21: state definitions. For example, lines 7–11 define state `s0`. Note that this is an initial state.

22–37: transition definitions. For example, in Fig. 3.1, we see that there is a `TT` transition between states `s0` and `s1`, which is encoded in lines 23–25.

3.2 Algebras

The class of logics χChek can use are those whose logical values form a finite distributive lattice, and where there is a suitably defined negation operator that preserves De Morgan laws and involution ($\neg\neg a = a$). Such lattices are called quasi-boolean, and the resulting structures are called quasi-boolean algebras [Ras78].

3.2.1 Available algebras

χChek is distributed with some of the more commonly used multi-valued logics, like Boolean - for classic model checking [CGP99], Kleene - used for vacuity checking [GC04], 4-valued disagreements logic - used for reasoning about model disagreements [EC01], upset - used for query checking [GCD03], etc. For SMV and GCLang models, Boolean, Kleene and upset algebras are available. For XML models, the following algebras are available:

- `2val-logic.xml`: classic two-valued logic. Can be used for model checking.
- `3val-logic.xml`: Kleene three-valued logic {`T, M, F`} (`M` = maybe). This logic is useful for representing partial models. See Gurfinkel and Chechik [GC04] for an explanation of how this logic is used for vacuity detection.
- `4val-logic.xml`: a 4-valued logic {`TT, TF, FT, FF`} that has been used to model disagreements that arise when composing two models drawn from different sources [EC01].
- `4fto-logic.xml`: a 4-value finite total order `F` $\Rightarrow$ `MF` $\Rightarrow$ `MT` $\Rightarrow$ `T`. `MF` = maybe false, `MT` = maybe true.
- `5fto-logic.xml`: a 5-value finite total order `F` $\Rightarrow$ `PF` $\Rightarrow$ `PT` $\Rightarrow$ `MT` $\Rightarrow$ `T`. `PF` = possibly false, `PT` = possibly true.

Each of these algebras is encoded in an XML file, which maps the algebra's Hass diagram to a GXL graph (edges are the truth ordering). Thus, users can introduce their own logics, by encoding the corresponding Hass diagrams.

3.2.2 Encoding an algebra

An algebra encoding is divided into five parts:

- Namespace declaration.
- (optional) Link to image of the corresponding Hass diagram.
- Logic values: one `<node>` tag per logic value in the algebra. The `<node>` tag can have an optional `<attr>` tag, where a description of the logic value can be specified.
- Logic order: specify the ordering of the lattice through the *above* relationship. One `<edge>` tag per edge in the lattice.
- Negation order: describes the result of negating the different logic values through the *neg* relationship. Must specify the negation of all logic values (one `<edge>` tag per logic value).

Basic structure of an algebra file:

```
<gxl xmlns:xbel='www.cs.toronto.edu/xbel' xmlns:xlink='xlink'>
 <graph ID='user_defined_id' edgemode='directed'>

   <!-- OPTIONAL: Hass diagram image -->
   <xbel:img xlink:type='simple' xlink:href='link_to_image_of_Hass_diagram'/>

   <!-- LOGIC VALUES -->
   <node ID='logic_value'/>

   <node ID='logic_value'>
     <!-- OPTIONAL -->
     <attr name='desc' value='description'/>
   </node>

   <!-- LOGIC ORDER -->
   <!-- interpreted as: logic_value1 is above logic_value2 in the lattice -->
   <edge from='logic_value1' to='logic_value2'>
     <type value='above'/>
   </edge>

   <!-- NEGATION ORDER -->
   <!-- interpreted as: logic_value1 is the negation of logic_value2 -->
   <edge from='logic_value1' to='logic_value2'>
     <type value='neg'/>
   </edge>

 </graph>
</gxl>
```

EXAMPLE: 4-valued disagreements logic

Now we will show how to encode the logic shown in Fig. 3.2.

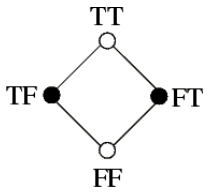

Figure 3.2: 4-valued disagreements logic

This is the encoding for this logic:

```
1  <gxl xmlns:xbel='www.cs.toronto.edu/xbel' xmlns:xlink='xlink'>
2   <graph ID='4-val' edgemode='directed'>
3     <xbel:img xlink:type='simple' xlink:href='examples/xml/4val.gif'/>
4     <!-- LOGIC VALUES -->
5     <node ID='TT'/>
6     <node ID='TF'/>
7     <node ID='FT'/>
8     <node ID='FF'/>
9     <!-- LOGIC ORDER -->
10    <edge from='TT' to='TF'>
11     <type value='above'/>
12    </edge>
13    <edge from='TT' to='FT'>
14     <type value='above'/>
15    </edge>
16    <edge from='TF' to='FF'>
17     <type value='above'/>
18    </edge>
19    <edge from='FT' to='FF'>
20     <type value='above'/>
21    </edge>
22    <!-- NEGATION ORDER -->
23    <edge from='TT' to='FF'>
24      <type value='neg'/>
25    </edge>
26    <edge from='FF' to='TT'>
27      <type value='neg'/>
28    </edge>
29    <edge from='FT' to='TF'>
30      <type value='neg'/>
31    </edge>
32    <edge from='TF' to='FT'>
33      <type value='neg'/>
34    </edge>
35   </graph>
36  </gxl>
```

An explanation of the various items in the logic encoding file:

1-2: GXL namespace declaration, the graph is directed and it's id is "4-val".

3: link to an image showing the logic's lattice. In this case, the image linked is the one shown in Fig. 3.2.

4–8: logic values `TT`, `TF`, `FT` and `FF`.

9–21: edges of the graph, describing the logic's truth ordering. For example, in Fig. 3.2, we see that `TT` is above `TF` in the lattice ordering, which is encoded in lines 10–12.

22–34: negation order, which describes the result of negating the different logic values. For example, lines 29–21 tell us that `FT` is the negation of `TF`.

3.3 Properties

Properties are specified in Computation Tree Logic (CTL) [CGP99]. The syntax of CTL formulas recognized by χChek is as follows:

```
ctlExpr ::
    basicExpr                           -- a simple boolean expression
    | '(' ctlExpr ')'
    | ctlExpr '|' ctlExpr               -- disjunction; alternative symbol: \/
    | ctlExpr '&' ctlExpr               -- conjunction; alternative symbol: /\
    | '!' ctlExpr                       -- negation; alternative symbol: ~
    | ctlExpr '->' ctlExpr              -- implication
    | ctlExpr '<->' ctlExpr             -- if and only if
    | 'EX' ctlExpr                      -- exists next state
    | 'AX' ctlExpr                      -- forall next state
    | 'EF' ctlExpr                      -- exists finally
    | 'AF' ctlExpr                      -- forall finally
    | 'EG' ctlExpr                      -- exists globally
    | 'AG' ctlExpr                      -- forall globally
    | 'E[' ctlExpr 'U' ctlExpr ']'      -- exists until
    | 'A[' ctlExpr 'U' ctlExpr ']'      -- forall until
    | 'E[' ctlExpr 'W' ctlExpr ']'      -- exists weak until
    | 'A[' ctlExpr 'W' ctlExpr ']'      -- forall weak until
    | 'E[' ctlExpr 'R' ctlExpr ']'      -- exists release
    | 'A[' ctlExpr 'R' ctlExpr ']'      -- forall release

basicExpr ::
    atomic
    | placeholder
    | '(' ctlExpr ')'

atomic ::
    identifier
    | number

placeholder ::
    '?' atomic '{' atomicSet '}'

atomicSet ::
    atomic ( ',' atomic )*

identifier ::
    ( alpha | '_' )( alpha | '_' | '-' | '.' | digit )*

number :: digit(digit)*

alpha :: 'a'..'z', 'A'..'Z'

digit :: '0'..'9'
```

Note: Usual precedence rules apply.

EXAMPLES:

- `EF (CC = Cruise & Throttle = ThrottleMaintain)`

  Model checking property. `CC` and `Throttle` are model variables in a flattened SMV model, and `Cruise` and `ThrottleMaintain` are possible values of these variables. See Section 5.1 for an example of how model checking is done in χChek.

- `AG (!connected -> (A[!connected U offhook]  AG !connected))`

Model checking property. `connected` and `offhook` are variables of a 4-valued model (logic shown in Fig. 3.2), i.e., these varibles can take the following values: `TT, TF, FT, FF`. See Section 5.1 for an example of how model checking is done in χChek.

- `AG (M & pc2 = 3 -> y1)`

  Using Kleene logic in the property for vacuity checking. This is a property for the GCLang version of the Peterson mutual exclusion algorithm given in Section 3.1.2. See Gurfinkel and Chechik [GC04] for an explanation of how Kleene logic is used for vacuity detection, and Section 5.2 for an example of how vacuity checking is done in χChek.

- `EF (?x{pc1,pc2} & EX (pc1 = 3 & pc2 = 3))`

  Query checking. This query is also w.r.t. the Peterson mutual exclusion algorithm. It has one placeholder – `?x`, which is restricted to the model variables `pc1, pc2`. In other words, χChek will look for the set of strongest propositional formulas, involving `pc1, pc2`, that make the query true. See [GCD03] for details on query checking, and Section 5.3 for an example of how query checking is done in χChek.

Chapter 4

Guide to the XChek User Interface

In this chapter, we show how various tasks are carried out in χChek. χChek is controlled through the main χChek GUI, shown in Fig. 4.1. There is no command-line interface.

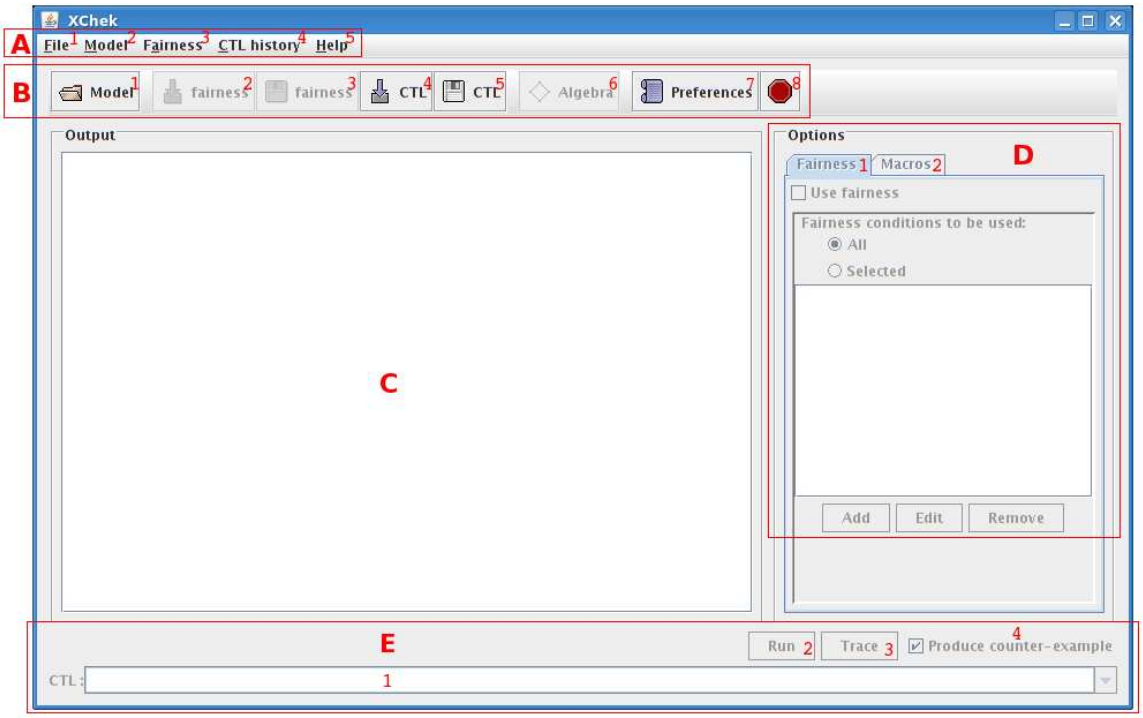

Figure 4.1: The main XChek windows.

Main window components:

A Menu

- **A1** File: access to *Model* options and *Preferences*.
- **A2** Model: access to *Model info* and *Show variables* functions. Not functional in this version.
- **A3** Fairness: load/save fairness constraints. Not functional in this version.
- **A4** CTL History: load/save CTL history.

A5 Help: access to *Help* and *About*.

B Quick access buttons

B1 Model: quick access to the *Model* options.

B2 (load) Fairness: load fairness constraints. Not functional in this version.

B3 (save) Fairness: save fairness constraints. Not functional in this version.

B4 (load) CTL: load CTL history. A CTL history file is just a simple list of properties (one per line). χChek creates a drop-down menu with these properties, which is accessible from the CTL input box.

B5 (save) CTL: save CTL history.

B6 Algebra: change model algebra. Not functional in this version.

B7 Preferences: quick access to *Preferences*.

B8 Quit: exits χChek. If there are properties in the CTL input box, χChek will prompt the user, asking if these properties should be saved.

C Main output area: various system messages appear in this area, e.g., success on model loading, results of model checking, etc.

D Options: these options are not functional in this version.

D1 Fairness: will allow the specification of fairness constraints, to be applied to the currently loaded model.

D2 Macros: will allow the specification of macros, which can be used to simplify CTL properties and fairness constraints.

E CTL area

E1 Input box: properties can be typed directly into this box. Can also be used to access the drop-down menu created when loading a CTL history file.

E2 Run: runs χChek, checking the property selected in the input box w.r.t. the last model loaded. Results will be displayed in the main output area (C).

E3 Trace: for simulation purposes. Not functional in this version.

E4 Produce counter-example: when this box is ticked, *Run* will also produce a counterexample for the property being checked. The manner in which this counterexample is shown depends on the user's preferences.

4.1 Loading a model

To load a model, press the *Model* button. The *Pick a compiler* window should appear (see Fig. 4.2).

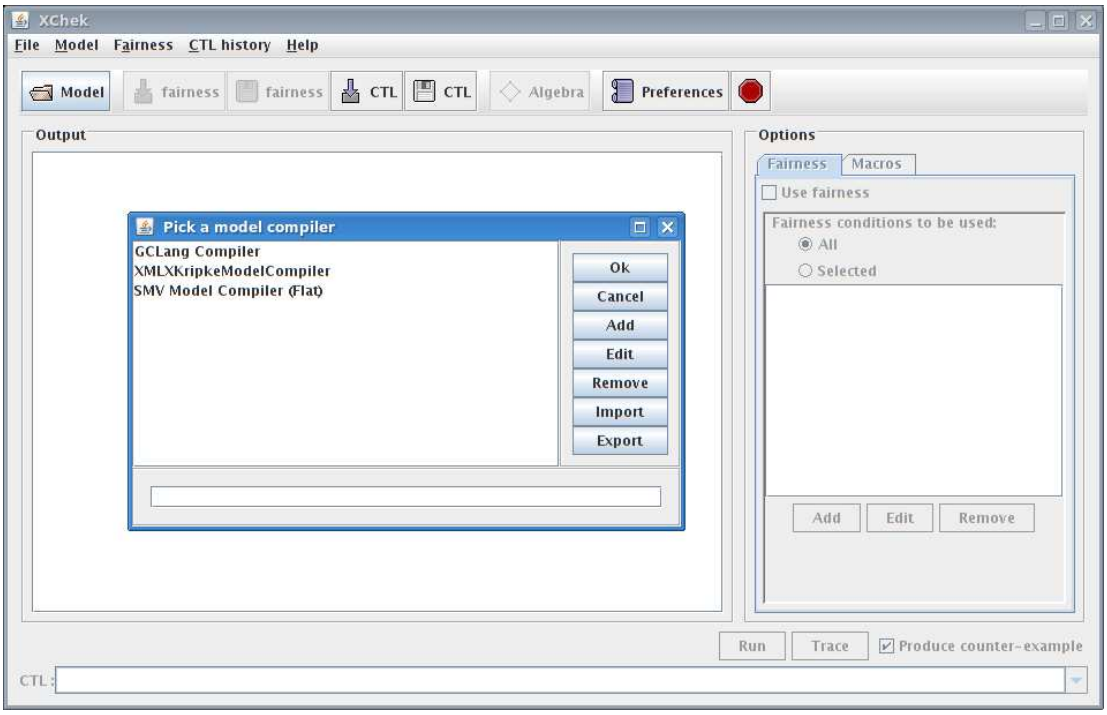

Figure 4.2: The *Pick a compiler* window.

As discussed in Chapter 3, χChek supports SMV, GCLang and XML as input formats for models. Thus, the three corresponding compilers appear in this window.

4.1.1 SMV and GCLang models

SMV and GCLang models are loaded in the same way (the only difference is the compiler). Thus, in this section, we only show the steps for loading an SMV model.

To load a flat SMV model, choose the *SMV Model Compiler (Flat)* compiler in the *Pick a compiler* window. An *Options* window will appear (see Fig. 4.3. For SMV models, there are three options: *Model Checking Algebra*, *SMV File* and *MvSet Implementation*.

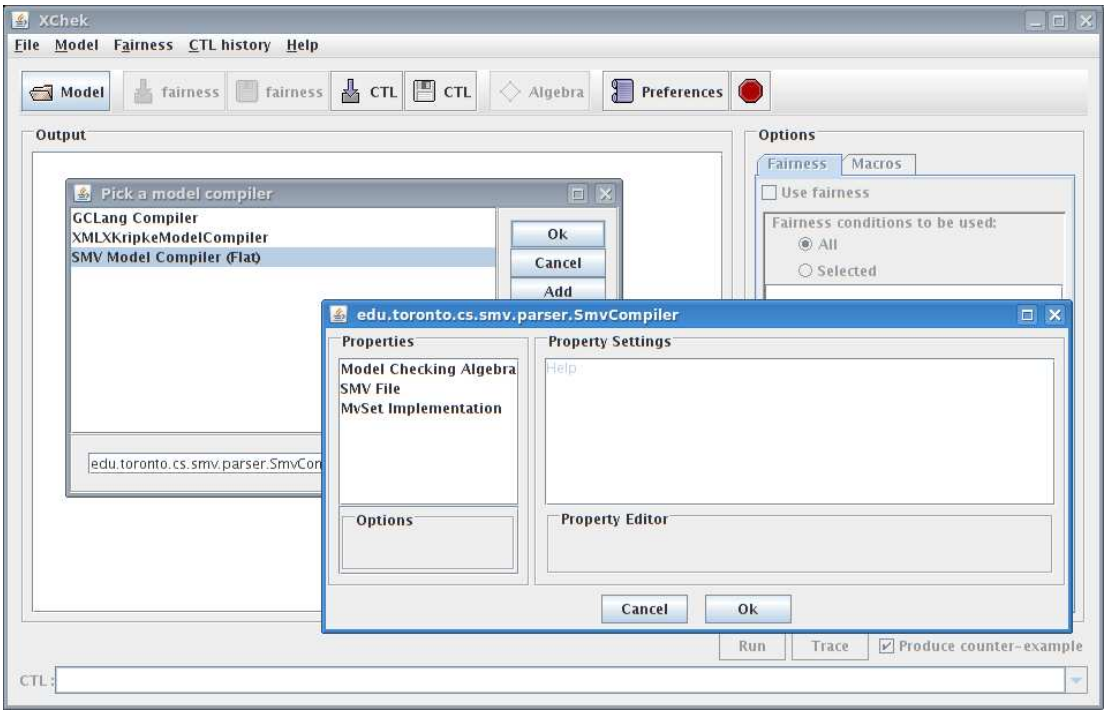

Figure 4.3: The *Options* window for the flat SMV model compiler.

Click on an option to set its value:

- Model Checking Algebra: drop-down list of possible logics. User picks a logic for the model that is currently being loaded. This choice depends on what χChek is being used for. For:
  - classical 2-valued model checking, pick *2*,
  - vacuity checking according to Gurfinkel and Chechik [GC04], pick *Kleene*,
  - query checking according to Gurfinkel et al. [GCD03], pick *upset*.
- SMV File: use the *browse* button to pick a flat SMV model.
- MvSet Implementation: decision diagram implementation used for this model. We recommend using *mdd*.
  - *mdd*: our own multi-valued decision diagrams. Everything should work with this.
  - *jadd*: our own multi-valued decision diagrams over Boolean variables, used for debugging, testing and comparisons to CUDD [Som01]. Counterexamples may not always work with this.
  - *cudd-add*: CUDD-based implementation. Supports multi-valued analysis, but not counterexample generation.
  - *jcudd*: new CUDD-based implementation, developed for Yasm [GC05].

Once all the options have been set, press the *OK* button. χChek will produce the following message in the *main output area* if model loading was successful:

```
Compiled in xx.xxx s
 loaded
```

4.1.2 XML models

As XML models specify the logic used to encode the model, there are fewer options when loading an XML model. To load an XML model, choose the *XMLXKripkeModelCompile* compiler in the *Pick a compiler* window. The *Options* window for the XML compiler is shown in Fig. 4.4.

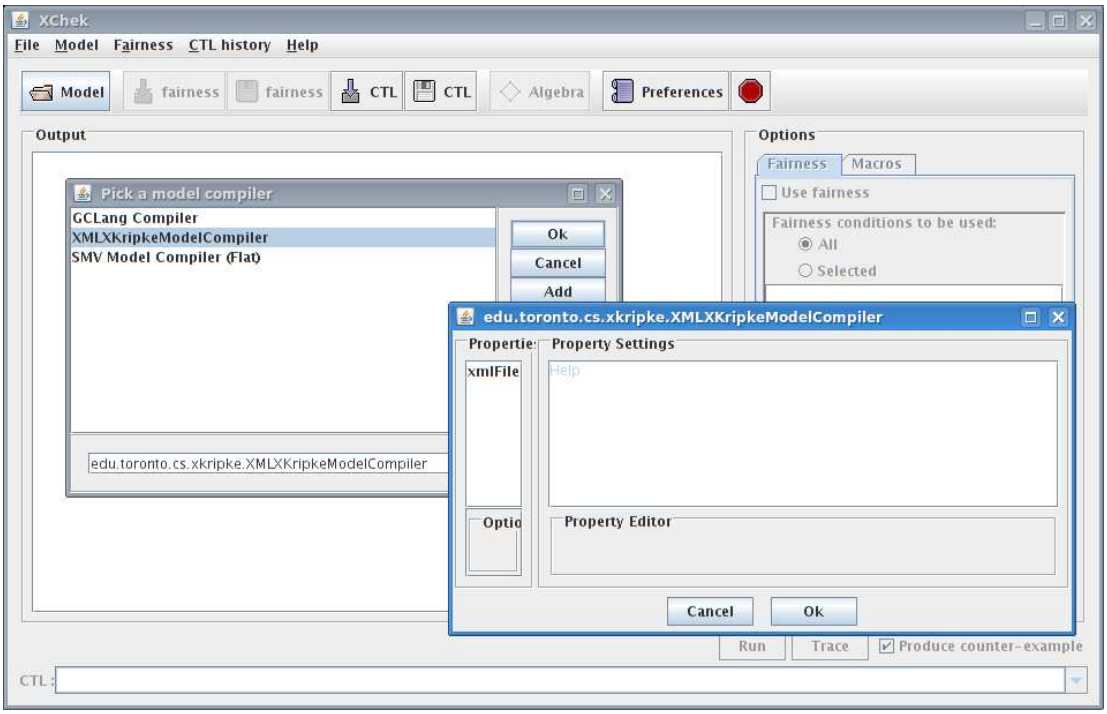

Figure 4.4: The *Options* window for the XML model compiler.

Only one option is available: *XML file*. Click on this option and use the *browse* button to pick an XML model. Afterwards, press the *OK* button. χChek will produce the following message in the *main output area* if model loading was successful:

```
Compiled in xx.xxx s
 loaded
```

4.2 CTL History files

A CTL history file is just a simple list of properties (one per line). To load a CTL history file, press the *(load) CTL* button. The user will be prompted for a history file. χChek will create a drop-down menu using this history file, which is accessible from the CTL input box.

For example, given the following CTL history file:

```
AG (state = Searching -> EF(state=Ready))
EF (state=Canceled)
AG AF(state=Canceled)
AG AF (state = Canceling)
```

χChek produces the drop-down menu seen in Fig. 4.5

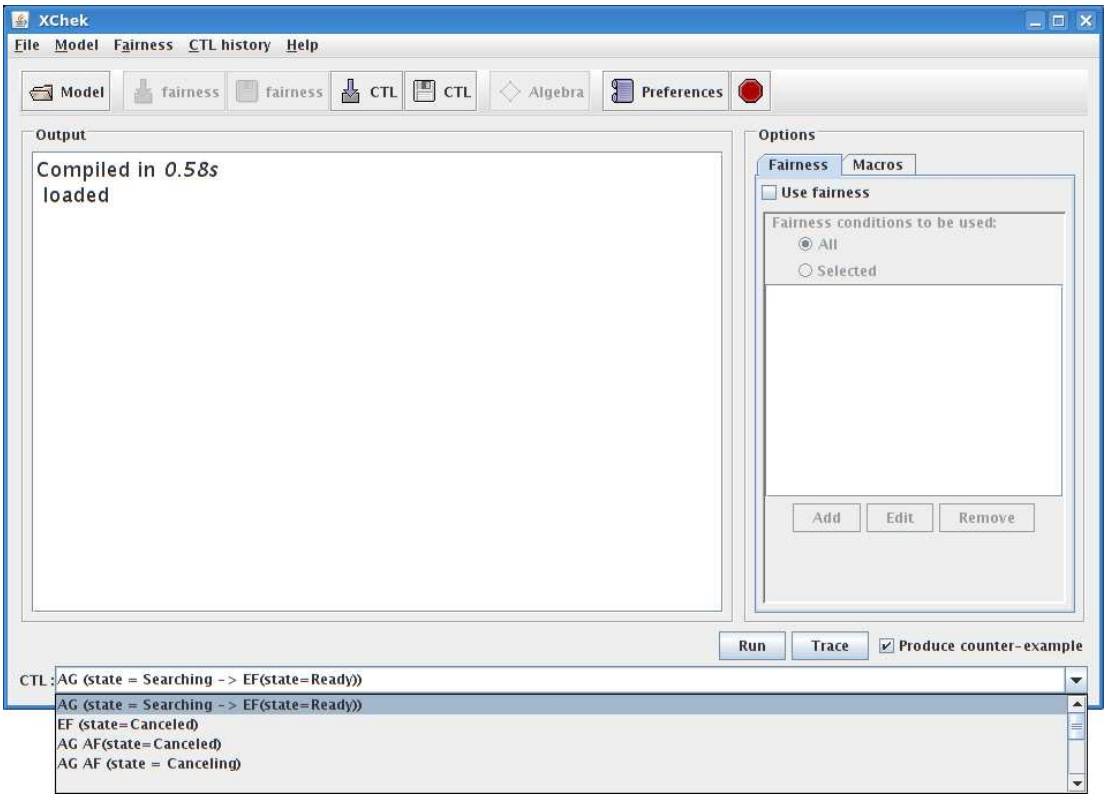

Figure 4.5: Drop-down menu created by loading a CTL history file.

At any moment, new properties can be added to the drop-down menu by typing them into the *CTL input box*. A list of properties can be saved to disk by pressing the *(save) CTL* button. The user will be prompted for the location of the save file.

4.3 Counterexamples

If a property is checked when the *Produce counter-example* box is ticked, χChek will produce a counterexample (or witness) for the property. This counterexample will be shown using one of the available counterexample viewers, which permit the interactive exploration and visualization of counterexamples. These counterexamples are generated using proof rules [GC03b, GC03a].

The default counterexample viewer (seen in Fig. 4.6) can show the counterexamples using different tools, with or without proofs. This viewer also offers a state-based view of the counterexample. When χChek is configured to use this counterexample viewer, proofs are generated prior to viewing. When a proof step is selected, the equivalent model state is shown. When the *Grappa* and *KegTree* preferences are set, the counterexample can be viewed using these tools, by pressing the *Grappa* or *DaVinci* buttons.

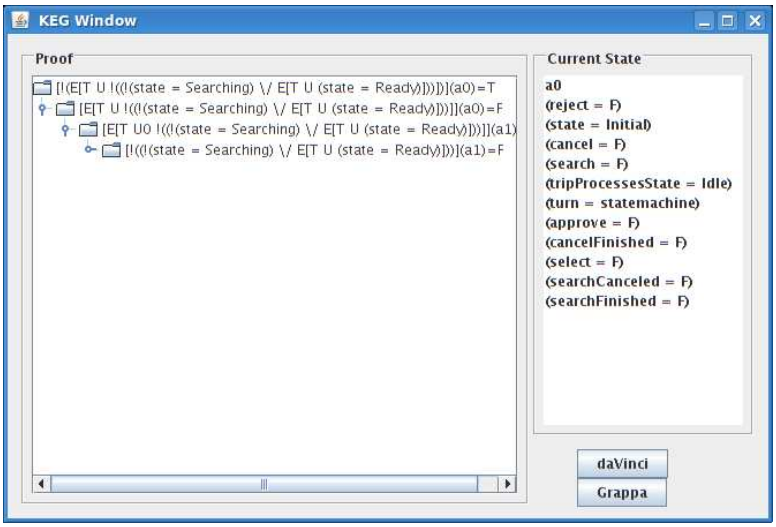

Figure 4.6: Default counterexample viewer.

The *Grappa* button generates the `dot` graph corresponding to the counterexample (see Fig. 4.7). The *DaVinci* button starts up uDraw(Graph) (see Fig. 4.8). uDraw(Graph) graphs are expandable – you can click on a node to see a subgraph. The *Hide Proof* option (see *KegTree* preferences) controls whether proof nodes of the counterexample should appear as expanded or not when the graph is initially presented.

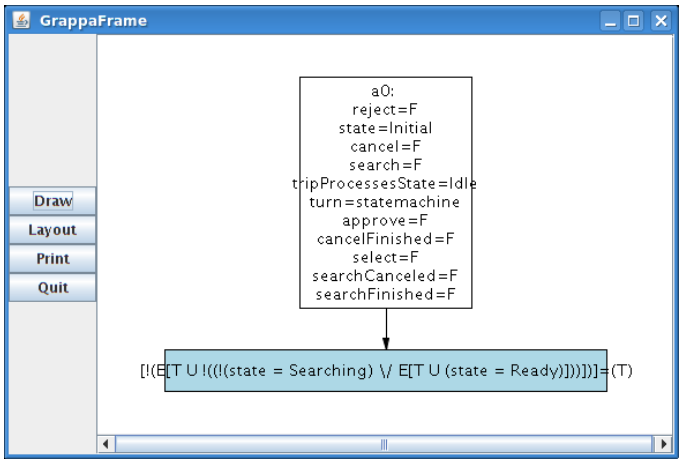

Figure 4.7: Counterexample view using Grappa.

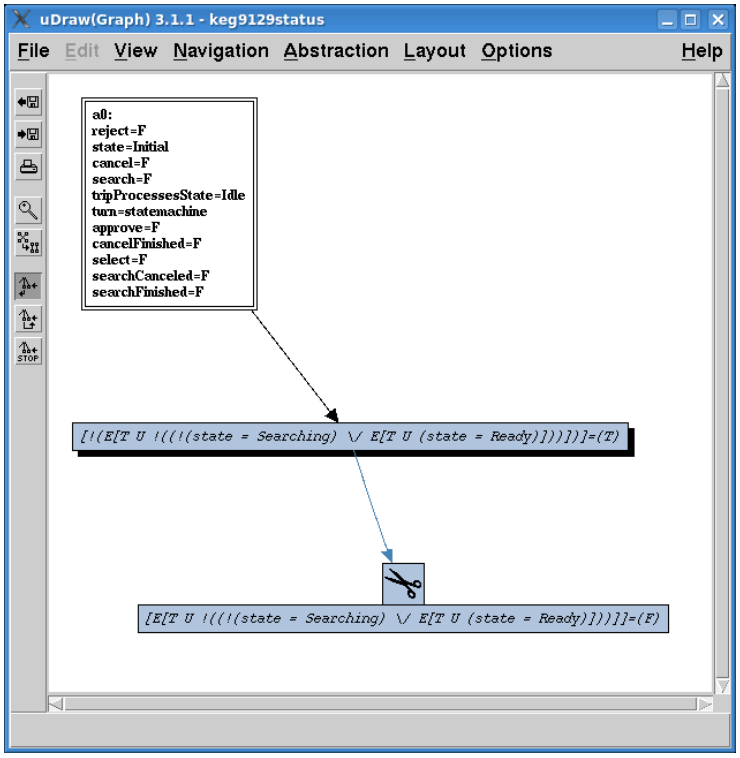

Figure 4.8: Counterexample view using uDraw(Graph).

The experimental counterexample viewer (seen in Fig. 4.9) is not as user-friendly as the default viewer, since this is a new feature. This viewer's interface is very simple and basically only displays the internal data strcture as is. However, it is interactive – the proof is generated as the user clicks on the leaves that she wants to expand.

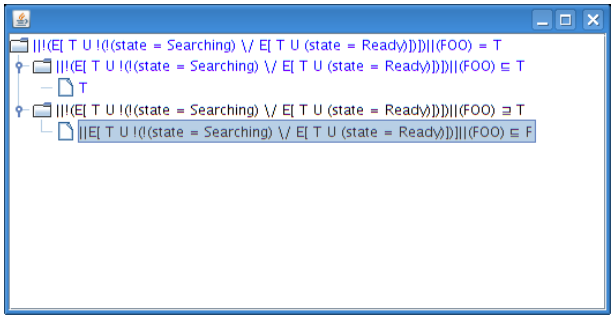

Figure 4.9: New, experimental counterexample viewer.

4.4 Preferences

To specify user preferences for χChek, press the *Preferences* button. The *XChek Preferences* window should appear (see Fig. 4.10). By default, the *General* preference options will be displayed. *Grappa* and *KegTree* preferences can also be set (see Figs. 4.11 and 4.12, respectively). Use the *Apply* button to apply preference changes, and close

the *XChek Preferences* window using the *Close* button. The *Import* and *Export* buttons can be used to load/save preferences to/from an XML file. We are use Java Preferences API (`http://java.sun.com/j2se/1.4.2/docs/guide/lang/preferences.html`) to deal with user preferences.

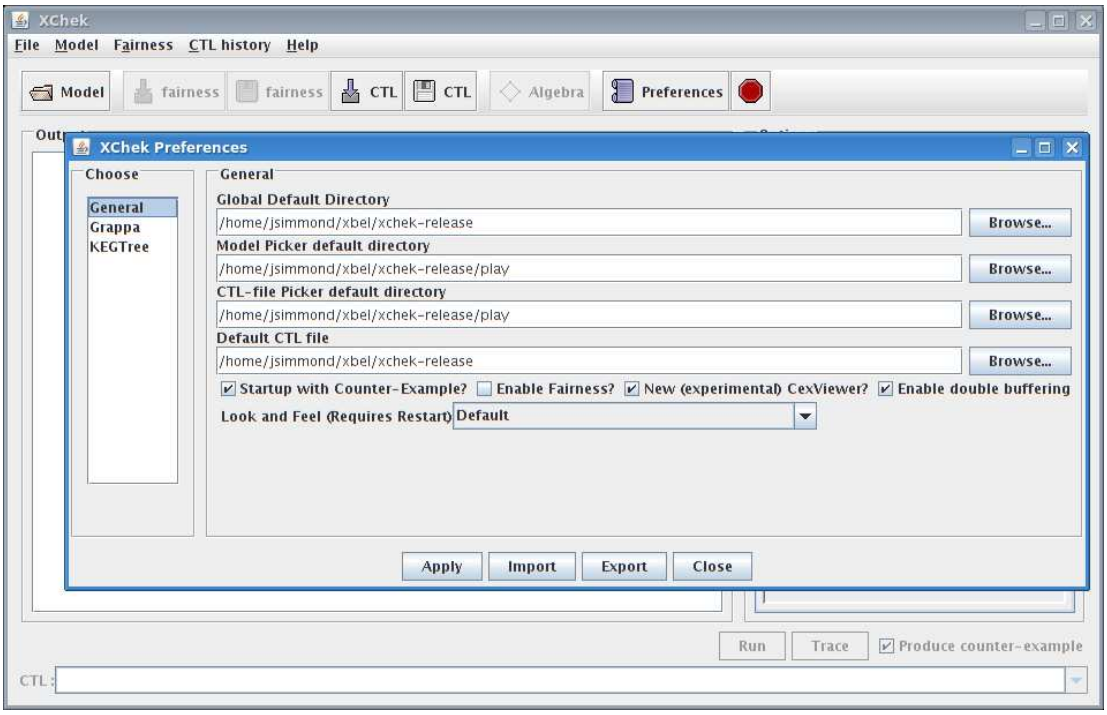

Figure 4.10: General preferences.

- *General*
  - Global default directory: use *Browse* button to change the value. This directory will be used as the base directory by χChek when loading files.
  - Model Picker default: use *Browse* button to change the value. This directory will be used as the base directory by χChek for loading model files.
  - CTL-file Picker: use *Browse* button to change the value. This directory will be used as the base directory by χChek for loading CTL history files.
  - Default CTL file: use *Browse* button to change the value. This CTL history file will be loaded by default when χChek is started.
  - Startup with Counter-Example: checking this box will ensure that the *Produce counter-example* box (E4) in the *Main* window is checked when χChek is started.
  - Enable Fairness: checking this box will ensure that the *Use fairness* box (D1) in the *Main* window is checked when χChek is started.
  - New (experimental) CexViewer: this option switches between the two available counterexample viewers. If checked, the new counterexample viewer will be used if the *Produce counter-example* box (E4) is checked in the *Main* window.
  - Enable double buffering (Remote X): checking this box will enable double buffering. This is recommended when running χChek remotely.

- *Grappa*: Grappa is a Java front-end to `dot`. Used to show counterexamples.
  - Path to DOT engine: use the *Browse* button to specify the path to `dot` executable. `dot` is used to make hierarchical or layered drawings of directed graphs, and is usually available by default in many linux distributions nowdays. If `dot` is not installed in your system, install the `graphviz` package.
- *KegTree*: KegTree is another way of presenting counterexamples. It produces a graph describing the counterexample in the format of the uDraw(Graph) tool.
  - Path to UDG home directory: use the *Browse* button to specify the path to your uDraw(Graph) home directory. uDraw(Graph) creates flow charts, diagrams, hierarchies or structure visualizations using automatic layout strategies. Available at `http://www.informatik.uni-bremen.de/uDrawGraph/en/index.html`
  - Hide proofs: controls whether proof nodes of the counterexample should appear as expanded or not when the graph is presented initially.

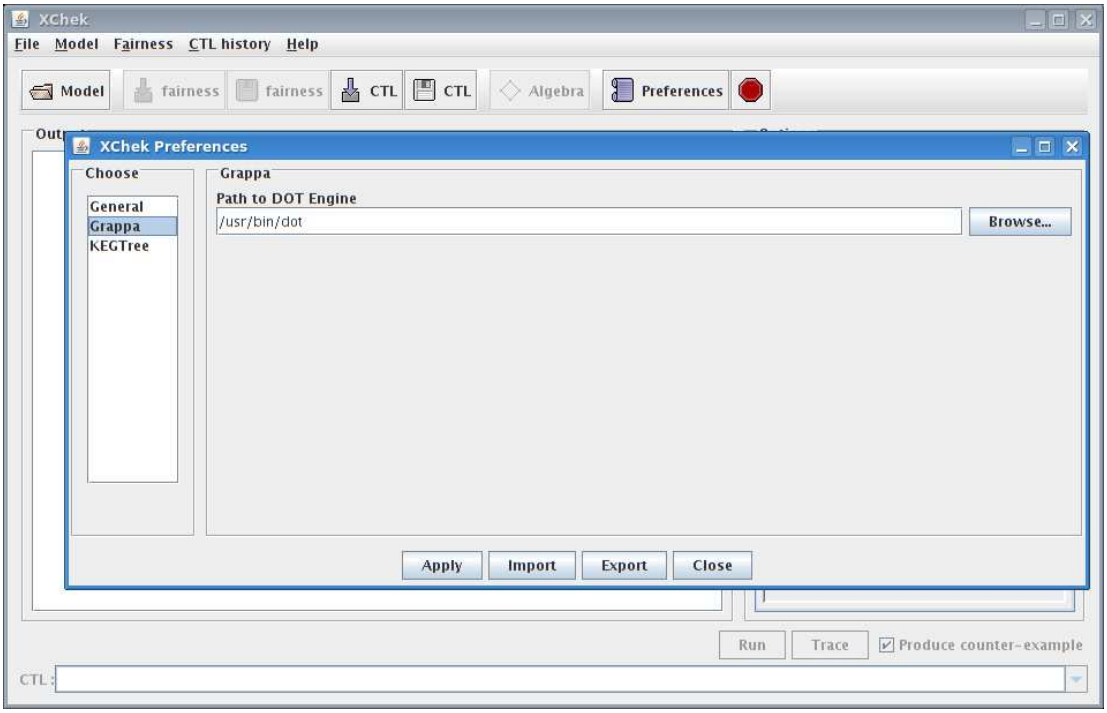

Figure 4.11: Grappa preferences.

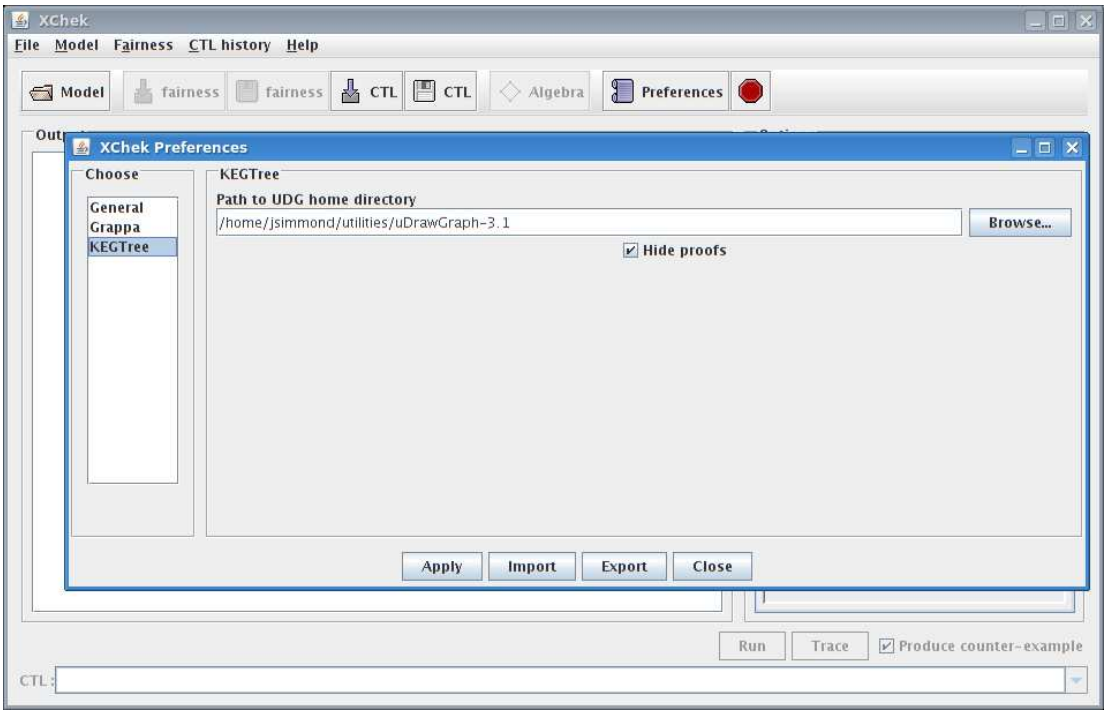

Figure 4.12: KegTree preferences.

Chapter 5

Tutorial

In this chapter, we show how χChek can be used for various purposes.

5.1 Model Checking

In the following example, we show how χChek can be used for classical 2-valued model checking.
Model: `/examples/gclang/trip/trip_planning5.gc`

1. *Main* window: press the *Model* button.
2. *Pick a compiler* window: choose the *GCLang Compiler*.
3. *Options* window:
   - Model Checking Algebra: *2*
   - GCLang file: `/examples/gclang/trip/trip_planning5.gc`
   - MvSet Implementation: *mdd*
4. *Options* window: press *OK* when options have been set. The *Main* window should look like the one in Fig. 5.1 if the steps uptil now have been followed correctly.
5. *CTL input box*: type in the following property
   `AG (state = Searching -> EF(state=Ready))`
6. (Optional) Check the *Produce counter-example* box to produce a counterexample for this property.
7. Press *Run*. The *Main* window should now look like the one in Fig. 5.2. If the *Produce counter-example* box is checked, the result is shown in a separate window, as described in Section 4.3.

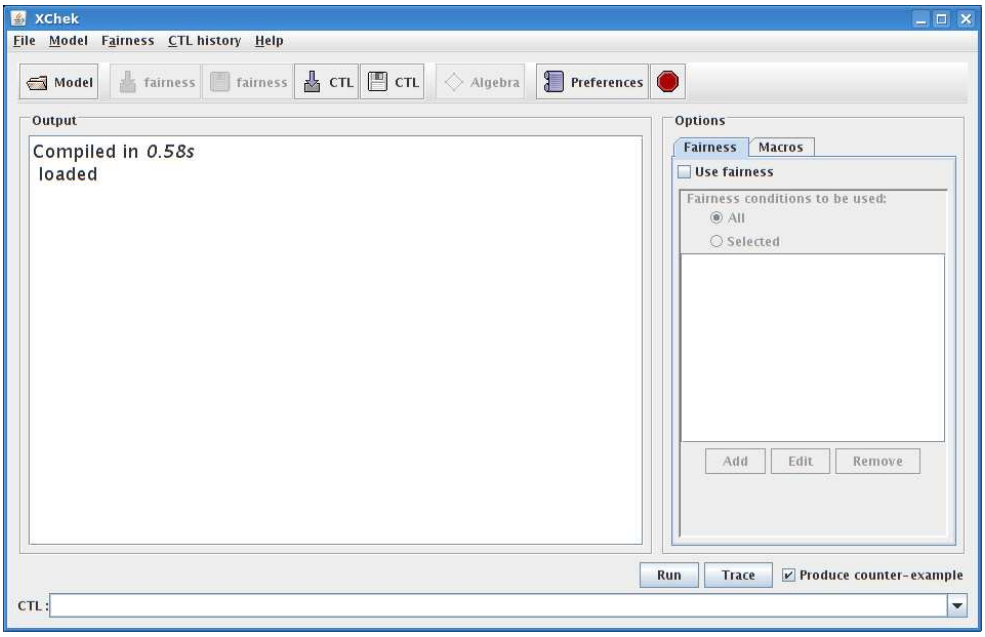

Figure 5.1: Model Checking: *main* window after successfully loading the model.

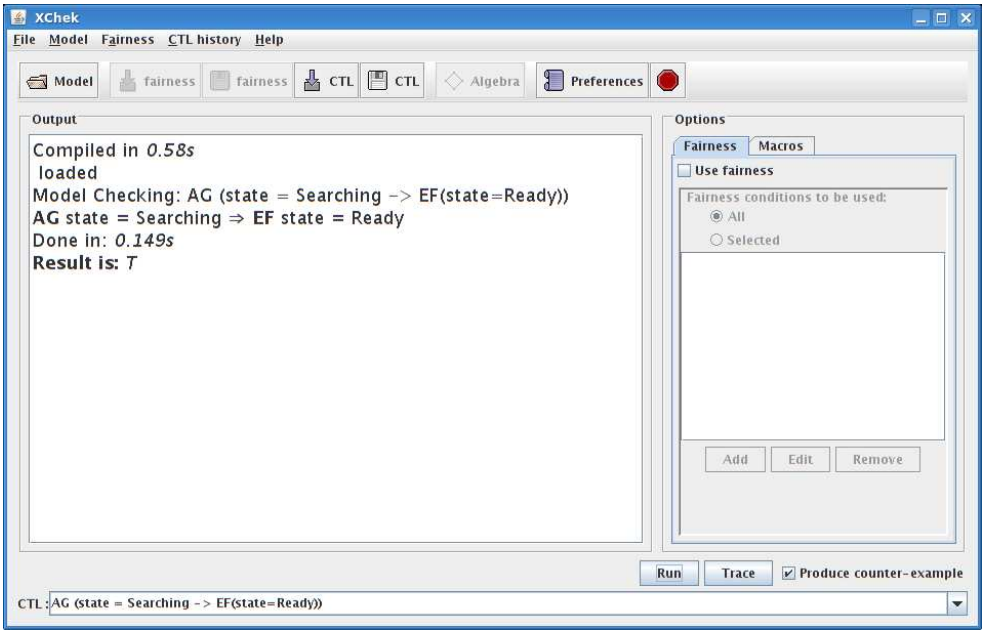

Figure 5.2: Model Checking: *main* window after successfully checking a property.

5.2 Vacuity Detection

In the following example, we show how χChek can be used for vacuity detection, according to See Gurfinkel and Chechik [GC04]. The 3-valued Kleene logic $\{F \Rightarrow M \Rightarrow T\}$ is used for this purpose. They show that is it possible to detect the vacuity of a propositional formula φ with respect to a formula ψ that is pure in φ, by (a) replacing all occurrences of ψ by M, and (b) interpreting the result in the 3-valued Kleene logic. If the new property evaluates to T, then φ is vacuously true. If it evaluates to F, then φ is vacuously false. Finally, if φ evaluates to M, no decision can be made w.r.t. the vacuity of the formula.

Model: `/examples/gclang/peterson.gc`

1. *Main* window: press the *Model* button.
2. *Pick a compiler* window: choose the *GCLang Compiler*.
3. *Options* window:
   - Model Checking Algebra: *Kleene*
   - GCLang file: `/examples/gclang/peterson.gc`
   - MvSet Implementation: *mdd*
4. *Options* window: press *OK* when options have been set.
5. *CTL input box*: type in the following property
   `AG (pc1 = 3 & pc2 = 3 -> y1)`
6. (Optional) Check the *Produce counter-example* box to produce a counterexample for this property.
7. Press *Run*. The *Main* window should now look like the one in Fig. 5.3. The property is true. Notice that we can check Boolean properties when the model is loaded as Kleene, since Boolean logic is subsumed by Kleene logic.
8. We will now check the vacuity of the formula, by checking the following formulas:
   `AG (M & pc2 = 3 -> y1)`
   `AG (pc1 = 3 & M -> y1)`
   `AG (pc1 = 3 & pc2 = 3 -> M)`

   CTL input box: type in each of these properties, press *Run* after each one. The *Main* window should now look like the one in Fig. 5.4 if the steps uptil now have been followed correctly.
9. Interpreting the results:
   `AG (M & pc2 = 3 -> y1)`, result: M
   `AG (pc1 = 3 & M -> y1)`, result: T
   `AG (pc1 = 3 & pc2 = 3 -> M)`, result: T

   The original property `AG (pc1 = 3 & pc2 = 3 -> y1)`, is vacuously true w.r.t. the following subformulas: `(pc2 = 3)`, `y1`.

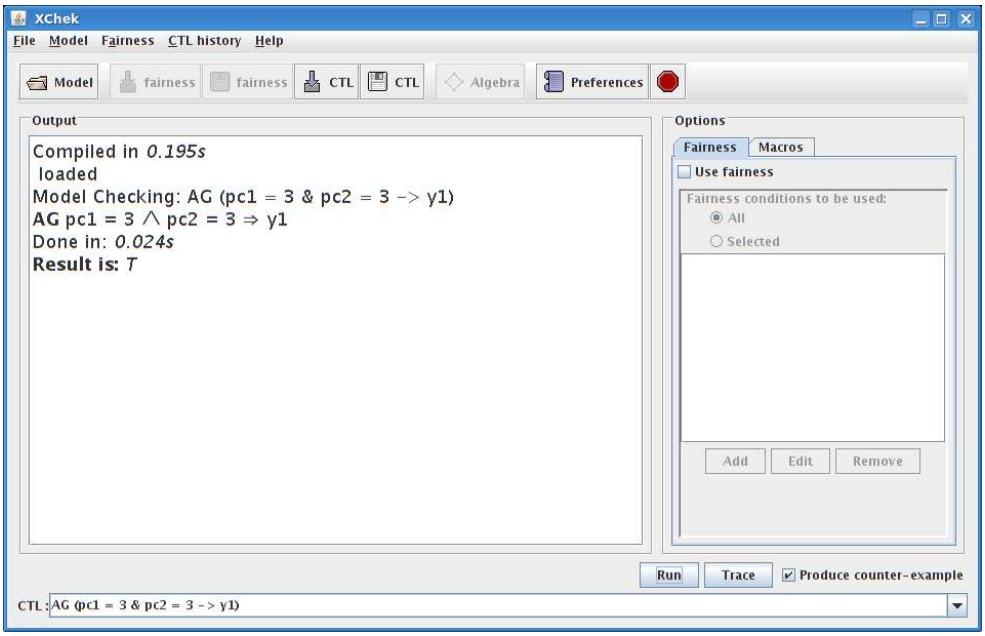

Figure 5.3: Vacuity Detection: *main* window after checking `AG (pc1 = 3 & pc2 = 3 -> y1)`.

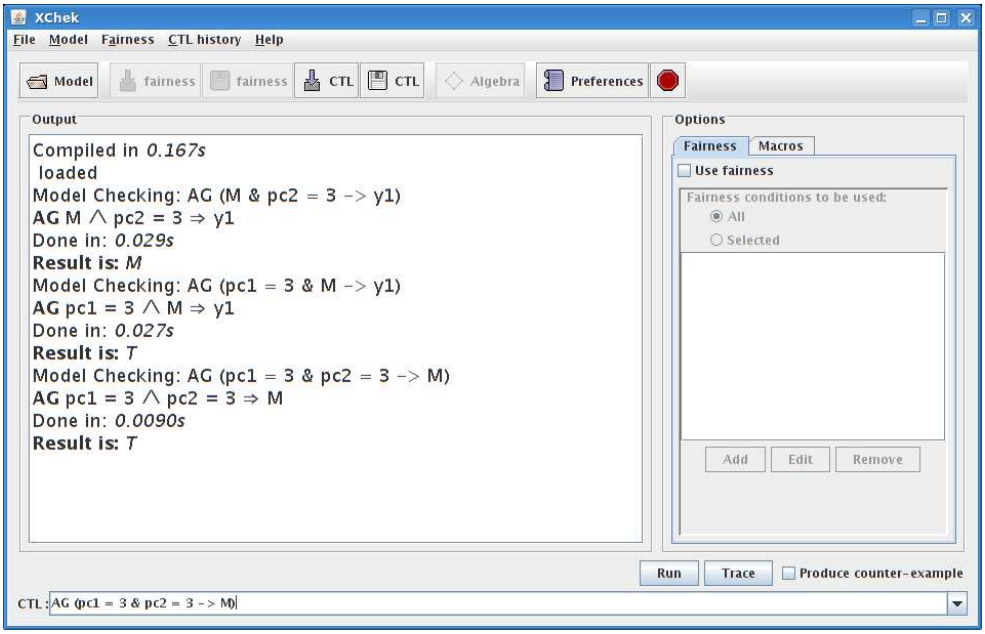

Figure 5.4: Vacuity Detection: *main* window showing vacuity detection results.

5.3 Query Checking

In the following example, we show how χChek can be used for query checking (see [GCD03]). Queries are expressed as CTL formulas with missing propositional subformulas, designated by placeholders ("?"). Query Checking finds the formulas that make the query true.

Model: `/examples/gclang/trip/trip_planning5.gc`

1. *Main* window: press the *Model* button.
2. *Pick a compiler* window: choose the *GCLang Compiler*.
3. *Options* window:
   - Model Checking Algebra: *upset*
   - GCLang file: `/examples/gclang/trip/trip_planning5.gc`
   - MvSet Implementation: *mdd*
4. *Options* window: press *OK* when options have been set.
5. *CTL input box*: type in the following property
   `AG (?x{state} -> EF(state=Ready))`
   This query has one placeholder, restricted to the `state` variable.
6. (Optional) Check the *Produce counter-example* box to produce a counterexample for this property.
7. Press *Run*. χChek will look for the set of strongest propositional formulas, involving `state`, that make the query true. The *Main* window should now look like the one in Fig. 5.5.
   There is one solution for this query:
   `state = Initial \/ state = Ready \/ state = Searching`

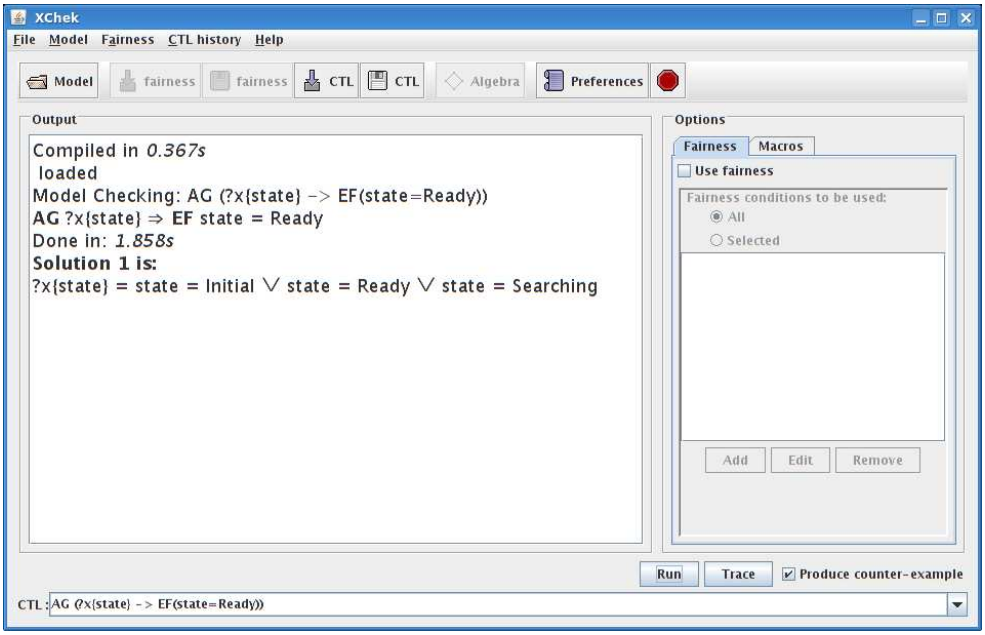

Figure 5.5: Query Checking: *main* window showing query checking results.

Bibliography

[CCG+02] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NUSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of CAV'02*, volume 2404 of *LNCS*, pages 359–364, 2002.

[CDEG03] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. Multi-Valued Symbolic Model-Checking. *ACM Transactions on Software Engineering and Methodology*, 12(4):1–38, October 2003.

[CDG02] M. Chechik, B. Devereux, and A. Gurfinkel. χChek: A Multi-Valued Model-Checker. In *Proceedings of 14th International Conference on Computer-Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 505–509, Copenhagen, Denmark, July 2002. Springer.

[CGD+06] M. Chechik, A. Gurfinkel, B. Devereux, A. Lai, and S. Easterbrook. "Symbolic Data Structures for Multi-Valued Model-Checking". *Formal Methods in System Design*, 29(3), 2006.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[EC01] S. Easterbrook and M. Chechik. "A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints". In *Proceedings of International Conference on Software Engineering (ICSE'01)*, pages 411–420, Toronto, Canada, May 2001. IEEE Computer Society Press.

[GC03a] A. Gurfinkel and M. Chechik. Generating Counterexamples for Multi-Valued Model-Checking. In *Proceedings of Formal Methods Europe (FME'03)*, volume 2805 of *LNCS*, pages 503–521, Pisa, Italy, September 2003. Springer.

[GC03b] A. Gurfinkel and M. Chechik. Proof-like Counterexamples. In *Proceedings of 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 160–175, Warsaw, Poland, April 2003. Springer.

[GC04] A. Gurfinkel and M. Chechik. How Vacuous Is Vacuous? In *Proceedings of TACAS'04*, volume 2988 of *LNCS*, pages 451–466, March 2004.

[GC05] A. Gurfinkel and M. Chechik. YASM: Model-Checking Software with Belnap Logic. Technical Report 533, University of Toronto, April 2005.

[GCD03] A. Gurfinkel, M. Chechik, and B. Devereux. Temporal Logic Query Checking: A Tool for Model Exploration. *IEEE Transactions on Software Engineering*, 29(10):898–914, October 2003.

[Gur02] A. Gurfinkel. Multi-Valued Symbolic Model-Checking: Fairness, Counter-Examples, Running Time. Master's thesis, University of Toronto, Department of Computer Science, October 2002.

[Ras78] H. Rasiowa. *An Algebraic Approach to Non-Classical Logics. Studies in Logic and the Foundations of Mathematics*. Amsterdam: North-Holland, 1978.

[Som01] F. Somenzi. "CUDD: CU Decision Diagram Package Release", 2001.