An Implementation of the Context-Based Tableau

Jose Gaintzarain Jon Ander Hernández Paqui Lucio

University of the Basque Country

1 Motivation
- Tableaus
- Rules of Propositional Tableaus

2 PLTL
- PLTL operators
- Rules of PLTL Tableaus
- One Pass vs Two Pass rules

3 Some TTM Implementation Details
- Loop detection
- The TTM Working Set
- PLTL Comparisons
- PLTL Hashing

4 TTM Architecture
- Alex
- Happy
- Gtk2Hs
- Graphviz
- Screenshot

5 Future Work

Motivation

Motivation

- Temporal logic plays a significant role in computer science, for reasoning about dynamic systems whose states change along the time.
- Tableau systems are refutational proof methods that serve as decision procedures for decidable logics.
- Propositional Linear Temporal Logic (PLTL) is decidable, but its worst-case is PSPACE.
- Since 1983 some tableau-based decision sytems for PLTL has been proposed, implemented and compared.

PLTL Tableaus

1. Wolper's Tableau (1983) for PLTL
   - EXPTIME (in the worst-case)
   - two-pass
     1.1 First-pass generates an auxiliary graph .
     1.2 Second pass checks and prunes the graph.

   Second pass is needed to check the satisfaction of the so-called eventualities.

2. Schwendimann's tableau (1998)
   - 2EXPTIME (in the worst-case)
   - is one-pass, it checks eventualities on-the-fly and branch-by-branch.

1. In PROLE 2007 (J. Gaintzarain et al.), the context-based tableau method was proposed, this method is one-pass without on-the-fly checking of eventualities.
   - Mechanism to force the fulfillment of eventualities (contradiction, when such fulfillment is imposible).
   - 2EXPTIME worst-case

The aim of this work (in progress) is to construct an efficient practical system that implements the context-based tableau method

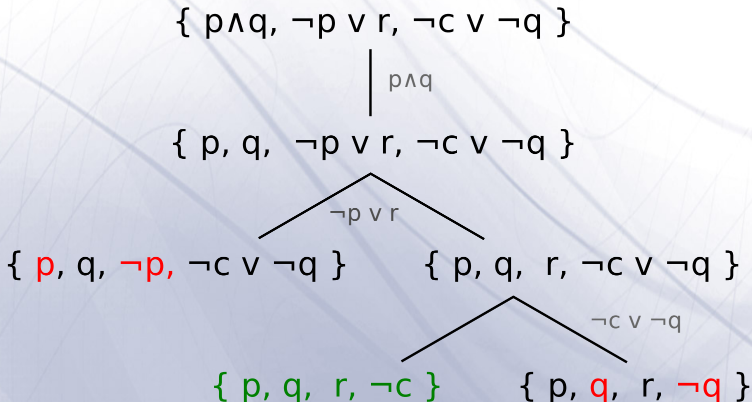
Introduction

Tableaus

- A tableau method for $\Delta \cup \{\neg\alpha\}$ obtains either :

  1. An Open Tableau. A countermodel of $\Delta \vdash \alpha$ or
  2. A Closed Tableau. A refutational proof of $\Delta \vdash \alpha$

- The main principle is to attempt to "break" complex formulae into smaller ones until complementary pairs of literals are produced or no further expansion is possible.
- If any branch of a tableau leads to an evident contradiction, the branch is defined as closed.
- If no further expansion is possible then the branch is defined as open.
- If the systematic tableau for ϕ is open, the open branch yields a model of ϕ.

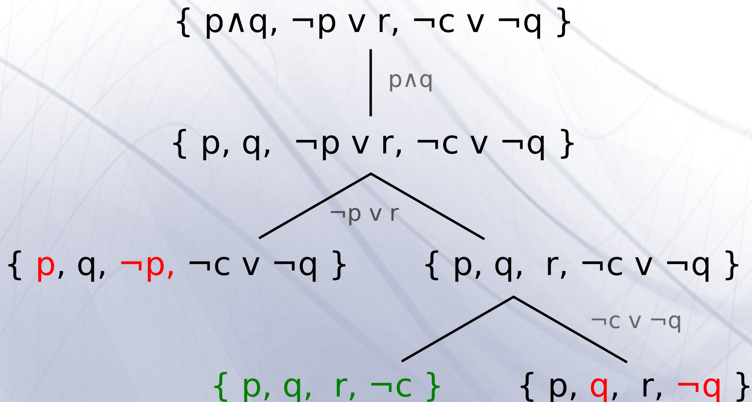

{ p∧q, ¬p v r, ¬c v ¬q }
p∧q
{ p, q, ¬p v r, ¬c v ¬q }
¬p v r
{ p, q, ¬p, ¬c v ¬q }
{ p, q, r, ¬c v ¬q }
¬c v ¬q
{ p, q, r, ¬c }
{ p, q, r, ¬q }

Propositional Rules

Rule	α	$A(\alpha)$
$(\neg\neg)$	$\neg\neg\varphi$	$\{\varphi\}$
$(\wedge)$	$\varphi \wedge \psi$	$\{\varphi, \psi\}$

Rule	β	$B_1(\beta)$	$B_2(\beta)$
$(\neg\wedge)$	$\neg(\varphi \wedge \psi)$	$\{\neg\varphi\}$	$\{\neg\psi\}$

$$\frac{\Delta, \neg\neg\varphi}{\Delta, \varphi} \qquad \frac{\Delta, \varphi \wedge \psi}{\Delta, \varphi, \psi} \qquad \frac{\Delta, \neg(\varphi \wedge \psi)}{\Delta, \neg\varphi \qquad \Delta, \neg\psi}$$

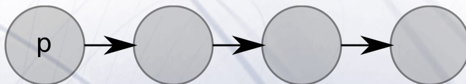

{ p∧q, ¬p v r, ¬c v ¬q }
p∧q
{ p, q, ¬p v r, ¬c v ¬q }
¬p v r
{ p, q, ¬p, ¬c v ¬q }
{ p, q, r, ¬c v ¬q }
¬c v ¬q
{ p, q, r, ¬c }
{ p, q, r, ¬q }

Propositional Linear Temporal Logic

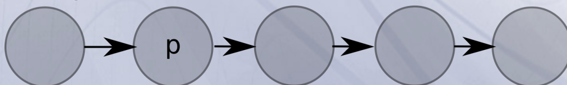
Propositional Linear Temporal Logic

- Propositional Operators
  - Atomic Propositions p, q, r, ...
  - Boolean Connectives $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$
- Temporal Operators
  - p "p is true now"
  - $\circ$p "p is true in the next state"
  - $\diamond$p "p will be eventually true in the future"
  - $\square$p "p will be always true in the future"
  - qUp "q will hold true until p eventually becomes true"
  - qRp "p always holds. A requirement that is released as soon as p holds"

1. p "p is true now"

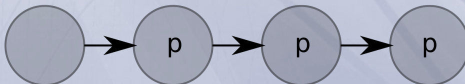


2. $\circ$p "p is true in the next state"

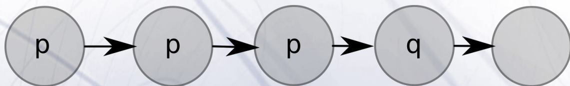


1. $\Diamond p$ "p will be eventually true in the future"

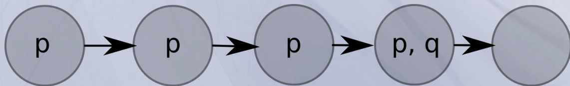
2. $\Box p$ "p will be always true in the future"



1. qUp "q will hold true until p eventually becomes true"



2. qRp "p always holds. A requirement that is released as soon as p holds"



PLTL Rules

Rule	α	$A(\alpha)$
$(\neg\neg)$	$\neg\neg\varphi$	$\{\varphi\}$
$(\wedge)$	$\varphi \wedge \psi$	$\{\varphi, \psi\}$
$(\neg\circ)$	$\neg\circ\varphi$	$\{\circ\neg\varphi\}$

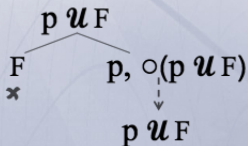
Rule	β	$B_1(\beta)$	$B_2(\beta)$
$(\neg\wedge)$	$\neg(\varphi \wedge \psi)$	$\{\neg\varphi\}$	$\{\neg\psi\}$
$(\neg\mathcal{U})$	$\neg(\varphi\mathcal{U}\psi)$	$\{\neg\varphi, \neg\psi\}$	$\{\varphi, \neg\psi, \neg\circ(\varphi\mathcal{U}\psi)\}$
$(\mathcal{U})_1$	$\varphi\mathcal{U}\psi$	$\{\psi\}$	$\{\varphi, \neg\psi, \circ(\varphi\mathcal{U}\psi)\}$

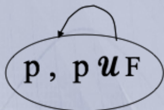
Rule	β	$B_1(\beta)$	$B_2(\beta, \Delta)$
$(\mathcal{U})_2$	$\varphi\mathcal{U}\psi$	$\{\psi\}$	$\{\varphi, \neg\psi, \circ((\varphi \wedge \Delta^{\neg})\mathcal{U}\psi)\}$ where Δ stands for the context

PLTL One Pass and Two Pass

Two-pass tableau

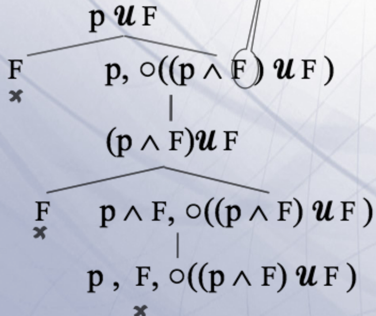
$$\frac{\varphi\ \mathcal{U}\ \psi}{\psi \qquad \varphi, \circ(\varphi\ \mathcal{U}\ \psi)}$$





One-pass tableau

$$\frac{\Delta, \varphi\ \mathcal{U}\ \psi}{\Delta, \psi \qquad \Delta, \varphi, \circ((\varphi \wedge \neg\Delta)\ \mathcal{U}\ \psi)}$$



Some PLTL Tableau properties

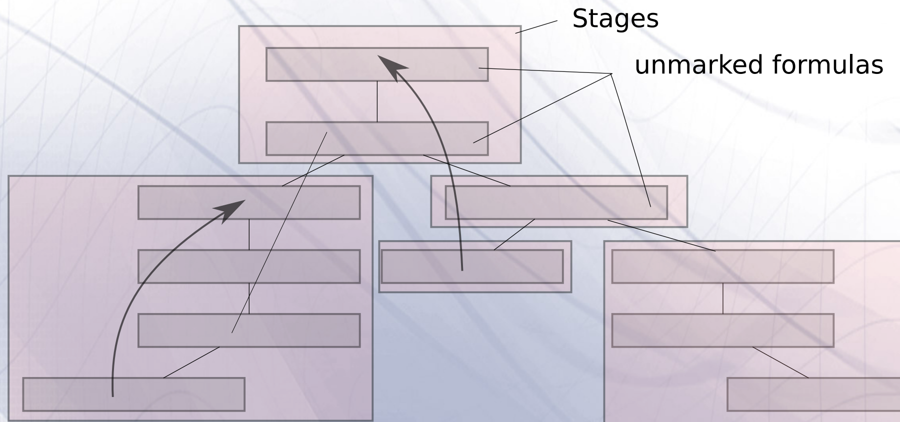
1. A branch can form a loop. For example:
   $path(b) = n_1, n_2, n_3 \langle n_4, n_5 \rangle^w \; L(n_5) = L(n_3)$
2. If $\{\phi, \neg\phi\} \subseteq L(s)$ for some stage s in a branch b, then every maximal branch prefixed by b is closed.
3. Weak Analytic superformula property (WASP)
   For every finite set of formulas, there exists a finite set that contains all the formulas that may occur in any systematic tableau.

Some TTM Implementation Details

```haskell
solveTableau :: Tableau -> Tableau
solveTableau []                = []
solveTableau (b:bs)            = solveBranch ++ solveTableau bs
    where   solveBranch        = if isCycling b then
                                        [b]
                                  else if consistentBranch b then
                                        solveTableau (↩
                                            extendBranch b)
                                      else []

extendBranch :: Branch -> Tableau
extendBranch b
    | nonElementary' == []     = [b ++ [unnext lastNode]]

    | isJust selection' &&
      (isNext . fromJust) selection'
                               = [b ++ [n] | n <- applyRules $ ↩
                                   fromJust selection' ]

    | otherwise                = [b ++ [n] | n <- applyRules $ ↩
        head nonElementary' ]

    where lastNode             = last b
          selection'           = selection              lastNode
          openFormulas'        = openFormulas           lastNode
          nonElementary'       = nonElementaryFormulas  lastNode
```

- TTM is implemented on Haskell
  1. Laziness vs Strictness
  2. Persistence and inmutable objects
  3. Hash Tables vs Balanced Trees
- Common operations
  - Adding/removing elements.
  - Node comparisons for detecting loops.
- Optimizations
  - Detection of repeated of unsatisfiable nodes.
  - Drop fullfilled eventualies

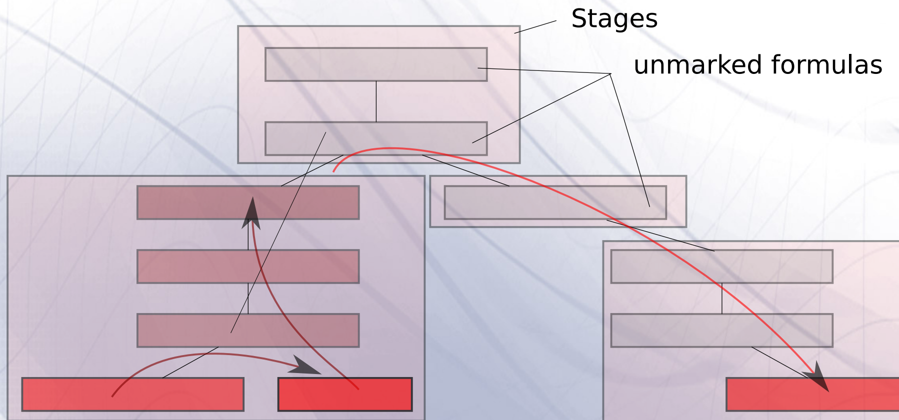
Loop detection



PLTL formula constructors

```haskell
data Node = Node {
    -- Guardamos el identificador del nodo (para saber dibujar
        el grafo)
    nodeId        :: !Int,

    -- Conjunto de formulas
    formulas      :: !Set,

    -- Hash
    nodeHash      :: !Int
}

instance Hashable Node where
    hash = nodeHash

instance Eq Node where
    a == b = (hash a) == (hash b) && (formulas a == formulas b
        )
```

Detection of repeated of unsatisfiable nodes



The working set

```
data WorkingSet = WorkingSet {
    -- Separate
    literalFormulas     :: !Set,
    -- Alfa formulas
    alwaysFormulas      :: !Set,
    alfaFormulas        :: !Set,
    -- Beta formulas
    eventuallyFormulas :: !Set,
    betaFormulas        :: !Set,
    -- Next formulas
    nextFormulas        :: !Set,
    -- Next always (for improving the negated contexts)
    nextAlwaysFormulas :: !Set,
    openFormulas        :: !Set,

    -- Already marked rules
    allFormulas         :: !Set,

    -- Fullfilling formulas
    fullfillEventuaMap :: !FullfillSet,
    fullfillReleaseMap :: !FullfillSet,

    inconsistent        :: Bool,

    setHash             :: !Int
```

PLTL formula constructors

```
-- Elemental formulas
trueFormula                 = TrueFormula        1
falseFormula                = FalseFormula       (-3)
variable    !q              = Variable q         (hash q)

pltlnot    !(Not p _)       = p
pltlnot    !(Variable q h)  = unary_hash Not (Variable q h) ↩
    (-991)
-- Sink the not
pltlnot     !q              = compl q

-- Avoid (always (always ...)
always      (Always q _)    = q
always      !q              = unary_hash Always q  (283)

-- Descompose (a implies b) into (not a v b)
implies     !q !p           = pltlor     (compl q) p

-- Binary operator
pltlor :: PLTLFormula -> PLTLFormula -> PLTLFormula
pltlor      !q !p           = binary_hash Or q p  (-113)
```

PLTL Comparisons

```haskell
instance Eq PLTLFormula where
    q == p = (hash q == hash p) && q === p

-- Simple PLTL Formulas
TrueFormula _    === TrueFormula _ = True
(Variable q _)   === (Variable p _) = q == p

-- Unary PLTL Formulas
(Always p _)     === (Always q _)   = (p === q)
(Not p _)        === (Not q _)      = (p === q)

-- Binary PLTL Formulas. Check the commutativity of formulas
(And q p _)      === (And r s _)     = ((q === r) && (p === s))↩
    ||
                                       ((q === s) && (p === r))

-- Check posible complementary formulas.
(Not (Variable p _) _)  === (Variable _ _) = False
(Not p _)               === q              = p === (compl q)
p                       === (Not q _)      = (compl p) === q

_                       === _              = False
```

PLTL Hashing

```
unary_hash !q !pri = (pri `combine` hash q)

binary_hash  !q !p !pri
                    = if (hash q) <= (hash p) then

                        pri `combine` hash q `combine` hash p

                      else

                        pri `combine` hash p `combine` hash q
```

Results

```
Random PLTL formulas with 12 operators
10,000 Random formulas
a: Different formulas :9753
b: Different hashes   :9152
a–b                   :601
100,000 Random formulas
a: Different formulas :89508
b: Different hashes   :73306
a–b                   :16202
250,000 Random formulas
a: Different formulas :207654
b: Different hashes   :155425
a–b                   :52229

real 0m38.422s
user 0m36.076s
sys 0m1.153s
```

```
unary_hash !q !pri =   (pri `combine` hash q `combine` hash q)

binary_hash !q !p !pri
                  = if (hash q) <= (hash p) then

                        pri `combine` hash q `combine` hash p
                            `combine` hash q `combine` 827

                    else

                        pri `combine` hash p `combine` hash q
                            `combine` hash p `combine` (-31)
```

```
Random PLTL formulas with 12 operators
    10,000 Random formulas
        a: Different formulas :9891
        b: Different hashesh  :9891
        a–b                   :0
    100,000 Random formulas
        a: Different formulas :93910
        b: Different hashes   :93887
        a–b                   :23
    250,000 Random formulas
        a: Different formulas :223801
        b: Different hashes   :223735
        a–b                   :66

real0m38.269s
user0m35.714s
sys 0m1.122s
```

TTM Architecture

Alex

- Alex is a tool for generating lexical analysers in Haskell, given a description of the tokens to be recognised in the form of regular expressions. It is similar to the tool lex or flex for C/C++.
- Tokens are defined as rules of a regular expressions
- Support Unicode.
- Alex 3.0 now does DFA minimization

- TTM uses Unicode symbols as the preferred way to represent PLTL operators.
- Classical PLTL operators are also supported (X, F, G, ...).
- Intended to support the grammar/tokens of other PLTL provers for future benchmarking.

```
%wrapper "basic"

-- Macros
$digit      = 0-9         -- digits
$alpha      = [a-zA-Z]    -- alphabetic characters

@ident      = $alpha[$alpha $digit \_ \']*
@not        = \x00AC | \- | \~
@or         = \x2228 | \v

tokens :-
<0> {
    @not                          { \s -> TNot       }
    @or                           { \s -> TOr        }
    "U"                           { \s -> TUntil     }
    @ident                        { \s -> TIdent s   }
    $white                        ;
}

{
data Token =  TIdent String   |
                TNot          |
                TAnd          |
                TOr           |
                TUntil
        deriving (Eq,Show)
```

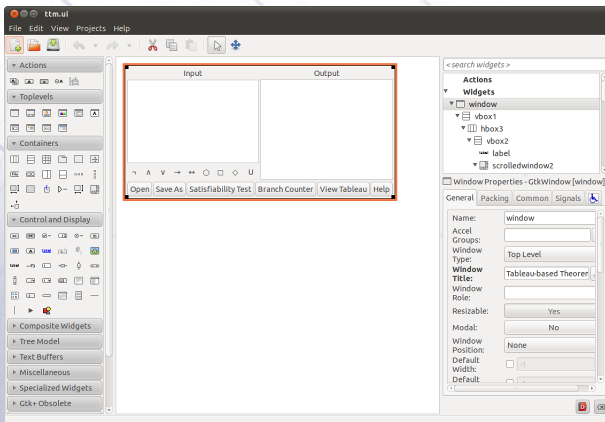
Happy

- Happy is a parser generator system for Haskell, similar to the tool 'yacc' for C. Like 'yacc',
- Happy files contain an annotated BNF specification of the grammar and produces a Haskell module
- Happy like Yacc generates a LALR parsers (by default).
  - LALR parsers are based on a finite-state-automata concept. The data structure used by an LALR parser is a pushdown automaton (PDA). A deterministic PDA is a deterministic-finite automaton (DFA) with the addition of a stack for a memory, indicating which states the parser has passed through to arrive at the current state.
  - GLR algorithm works in a manner similar to the LR parser algorithm, except that, given a particular grammar, a GLR parser will process all possible interpretations of a given input in a breadth-first search.

```
%name pltl
%tokentype { Token }
%error { parseError }
%token
    variable        { TIdent $$      }
    "not"           { TNot           }
    "or"            { TOr            }
    "until"         { TUntil         }
%%

Program:  Binary                              {Maybe $1         }
        | {- empty -}                         {Nothing          }

Binary:
          Binary2 "until" Binary2             {pltluntil $1 $3}
        | Binary2                             {          $1    }

Binary2:
          Binary2 "or" UnaryFormula           {pltlor    $1 $3}
        | UnaryFormula                        {          $1    }

UnaryFormula:
          "not" LiteralFormula                {pltlnot       $2}
        | LiteralFormula                      {              $1}

LiteralFormula:
          variable                            {variable      $1}
```

Gtk2hs

- Gtk2Hs is a GUI library for Haskell based on Gtk+. Gtk+ is an extensive and mature multi-platform toolkit for creating graphical user interfaces.
- Multiplataform.
- Glade. A RAD tool to enable quick & easy development of user interfaces for the GTK+.
- User interfaces are saved as XML files.


ttm.ui
File Edit View Projects Help
Actions
Toplevels
Containers
Control and Display
Composite Widgets
Tree Model
Text Buffers
Miscellaneous
Specialized Widgets
Gtk+ Obsolete
Input
Output
Open
Save As
Satisfiability Test
Branch Counter
View Tableau
Help
< search widgets >
Actions
Widgets
window
vbox1
hbox3
vbox2
label
scrolledwindow2
Window Properties - GtkWindow [window]
General
Packing
Common
Signals
Name:
window
Accel Groups:
Window Type:
Top Level
Window Title:
Tableau-based Theorem
Window Role:
Resizable:
Yes
Modal:
No
Window Position:
None
Default Width:
Default

```haskell
import Graphics.UI.Gtk

main = do
        initGUI

        -- Create the builder, and load the UI file
        builder <- builderNew
        builderAddFromFile builder "ttm.ui"

        -- Retrieve some objects from the UI
        window <- builderGetObject builder castToWindow   "window"
        button <- builderGetObject builder castToButton   "not"
        input  <- builderGetObject builder castToTextView "input"
        buf    <- textViewGetBuffer input

        -- Basic user interation
        button `onClicked` textBufferInsertAtCursor buf "\x00AC"
        window `onDestroy` mainQuit

        -- Display the window
        widgetShowAll window
        mainGUI
```

Graphviz

- Graphviz is open source graph visualization software.
- Descriptions of graphs is done in a simple text language (Dot language),
- Diagrams can be exported in useful formats: png image files and SVG for web pages, PDF or Postscript for inclusion in other documents;
- Allows options for colors, fonts, tabular node layouts, line styles, hyperlinks, rolland custom shapes.

Tableau-based Theorem Prover for Temporal Logic PLTL

Input

$\square$ a
$\square$ ($\bigcirc$ b)
$\square$ ($\bigcirc$ ($\bigcirc$ ($\bigcirc$ c)))

¬ ∧ ∨ → ↔ ○ □ ◇ U

Output

This is a model of Set:

State 0:{ a }
State 1:{ b, a }
State 2:{ b, a }
State 3:{ c, b, a }
Cycle starts at the state 3

Open | Save As | Satisfiability test | Branch Counter | View tableau | Help

Future Work

- Add Parallel and Concurrent support
- Add some heuristics or some queue with LRU behaviour to improve the list of unsatisfiable sets.
- Make a proper comparison with Logic Workbench[1]

[1] Implements Schwendimann's tableau