Theorem Proving beyond Deduction

Specification and Verification with Higher-Order Logic

Arnd Poetzsch-Heffter
(Slides by Jens Brandt)

Software Technology Group
Fachbereich Informatik
Technische Universität Kaiserslautern

Sommersemester 2008

Outline

Motivation

Unified Verification Platform

- use theorem prover to integrate various tools
- most general and most flexible tool

Combining Calculation and Deduction

- deduction: e.g. theorem proving:
  proving properties by mechanised logical deduction
- calculation: e.g. model checking:
  showing system $\mathscr{M}$ has property $\mathscr{P}$ by algorithmic calculation
- research goal:
  general platform for implementing provers and checkers

Want the Best of Both Worlds

want maximally expressive logics

- higher-order logic
- readable specs of whole systems, high-level datatypes etc.
- requires theorem proving

want state-of-the-art checking efficiency

- years of algorithm design and code honing

Three Approaches

Loose Integration

- link tools via protocols, scripting languages etc.

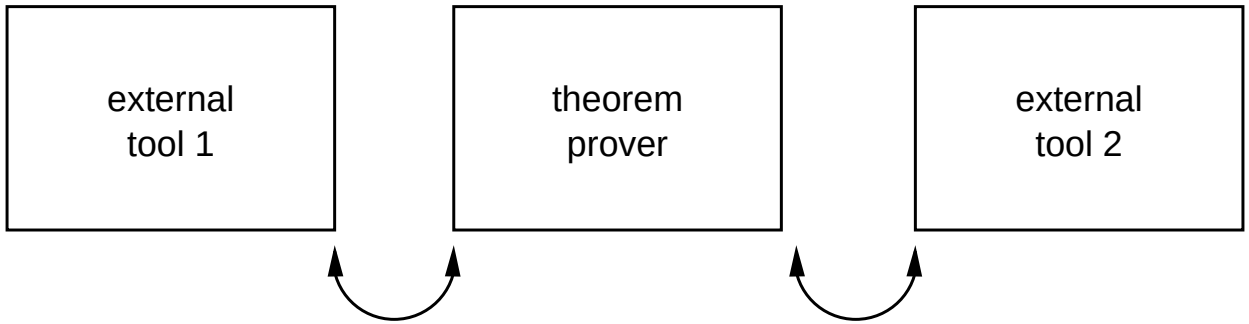
Add Calculation to Theorem Prover

- implement algorithms in a general theorem prover

Add Deduction to Specialised Tool

- add rules to a specialised tool

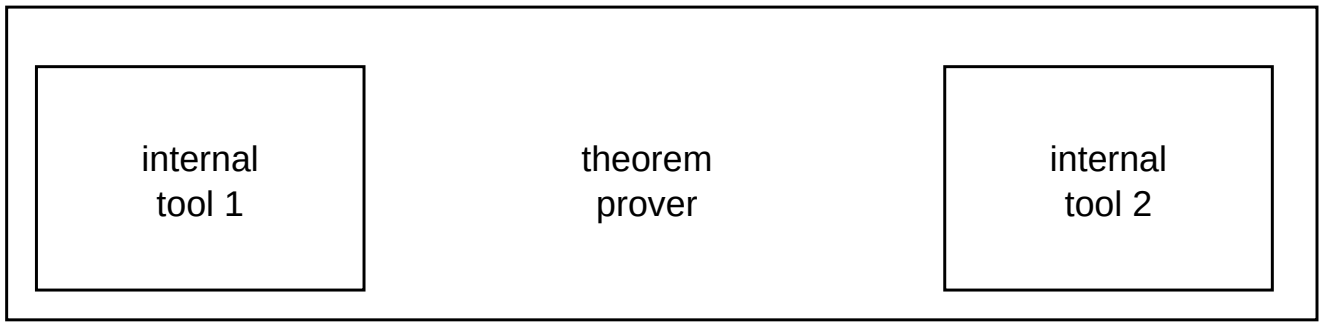
Loose Integration



Overview

- existing infrastructure, future-proof, semantically challenging
- many tools are linked in this way: SAT, FOL, ...
- approach followed by PROSPER EU project
- will be illustrated with the help of SAT solvers

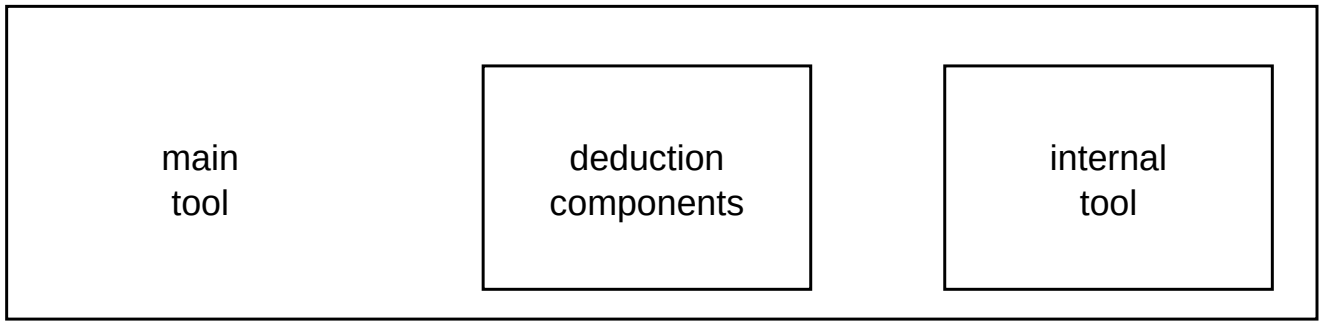
Add Calculation to Theorem Prover



Overview

- example: theorem proving is guided by model checking
- will be illustrated with the help of a model checker

Add Deduction to Specialised Tool



Overview

- goal: extend specialised tool to complete system
- lightweight proving + state-of-art checking (Cadence, Intel)
- will not be illustrated in the following

Outline

LCF Approach to Theorem Proving

Theorems

- theorems represented by an abstract type
- primitive operations
  - axioms
  - inference rules of a logic
- composing together the inference rules using ML programs

Example (Higher-Order Logic Axioms)

- $\vdash \forall\, b.\ (b = T) \vee (b = F)$
- $\vdash \forall\, b_1\ b_2.\ (b_1 \to b_2) \to (b_2 \to b_1) \to (b_1 = b_2)$
- $\vdash \forall\, f.\ (\lambda x.\ f x) = f$
- $\vdash \forall\, P\ x.\ P\ x \to P(\varepsilon\ P)$
- $\vdash \exists f.(\forall x\ y.\ f x = f y \to x = y) \wedge (\neg \forall x.\ \exists y.\ x = f\ y)$

LCF Approach to Theorem Proving

Backdoor

- oracles can create arbitrary theorems
- always tagged (string denoting the origin)
- tags are propagated
- if oracle is incorrect, all incorrect theorems can be spotted

Backdoor API

```
val mk_thm        : term list * term -> thm
val mk_oracle_thm : string -> term list * term -> thm
val add_tag       : tag * thm -> thm
val tag           : thm -> tag
```

Creating Tagged Theorems

Example (Oracles)

```
– show_tags := true;
> val it = () : unit

– val thm1 = mk_oracle_thm "me" ([],“T ==> F“);
> val thm1 = [oracles: me] [axioms: ] [] |– T ==> F : thm

– CONTRAPOS thm1;
> val it = [oracles: DISK_THM, me] [axioms: ] [] |– ~F ==> ~T : thm
```

Boolean Satisfiability Problem (SAT)

Problem

Given a Boolean expression, is there some assignment to all the variables that will make the entire expression true?

Example (SAT)

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

$$x_1 \wedge \neg x_2 \wedge (\neg x_1 \wedge x_2)$$

Properties

- problem is NP-complete
- very important, many applications
  - bounded model checking, equivalence checking, FPGA routing, ...

SAT Tool Interface: DIMACS

Interface

- common input file format for SAT solvers
- several descriptions supported
- in the following: expression in conjunctive form

Example (Input File Format)

$$(x_1 \vee x_3 \vee \neg x_4) \wedge x_4 \wedge (x_2 \vee \neg x_3)$$

```
c Example DIMACS file
p cnf 4 3
1 3 -4 0
4 0
2 -3 0
```

Using the SAT Library

Interface

- two functions that create theorems: satOracle and satProve
- difference: optional checking

Example (Using HolSatLib)

```
– satOracle grasp “(x ∨ ~y ∨ z) ∧ (~z ∨ y )“;
> val it = [oracles: grasp] [axioms: ] []
           |– z ∧ y ==> (x ∨ ~y ∨ z) ∧ (~z ∨ y) : thm

– satProve grasp “(x ∨ ~y ∨ z) ∧ (~z ∨ y )“;
> val it = [oracles: ] [axioms: ] []
           |– z ∧ y ==> (x ∨ ~y ∨ z) ∧ (~z ∨ y) : thm
```

Invoking the SAT Solver

Arguments of satOracle

- SAT solver: sato, grasp or zchaff ...
- term t

Steps of satOracle

- write a DIMACS format file corresponding to the term t
- invoke the solver on the file to create an output file
- parse the output file to extract the model found
- create a theorem (tagged with the name of the solver) that shows the model

Invoking the SAT Solver

Steps of satProve

- write a DIMACS format file corresponding to the term t
- invoke the solver on the file to create an output file
- parse the output file to extract the model found
- use HOL to check that the model is really a model and return an untagged theorem

Example (Using satProve)

```
– satProve grasp “(x ∨ ~y ∨ z) ∧ (~z ∨ y )“;
> val it = [oracles: ] [axioms: ] []
           |– z ∧ y ==> (x ∨ ~y ∨ z) ∧ (~z ∨ y) : thm
```

- checking a solution is relatively simple

Unsatisfiable Terms

Example (Unsatisfiable Terms)

```
– satOracle grasp "(x ∨ ~y ∨ z) ∧ ~z ∧ y ∧ ~x";
> val it = [oracles: grasp] [axioms: ] []
           |– ~((x ∨ ~y ∨ z) ∧ ~z ∧ y ∧ ~x)

– satProve grasp "(x ∨ ~y ∨ z) ∧ ~z ∧ y ∧ ~x";
! Uncaught exception:
! satProveError
```

Checking the Result

- proving that no solution exists is not simple
- no efficient implementation in HOL
  (basically needs to check all possibilities)

Tautology Checking

Procedure

1. prove $\vdash \forall \vec{x}.\ \neg t = \exists \vec{x}.\ t'$
2. use a SAT solver to prove $\vdash \exists \vec{x}.\ \neg t'$;
3. by negating both sides of (1), prove $\vdash \forall \vec{x}.\ \neg\neg t = \exists \vec{x}.\ \neg t'$
4. hence by combining (2) and (3) derive $\vdash \forall \vec{x}.\ \neg\neg t$.
5. hence by the law of double negation conclude $\vdash \forall \vec{x}.\ t$.

Tautology Checking

Example (Tautology Checking Code)

```
fun SAT_TAUT_CHECK sat_solver t =
 let val th1 = canonTools.CNF_CONV(mk_neg t)
     val th2 = satOracle sat_solver (rhs(concl th1))
     val th3 = AP_TERM "$~" th1
     val th4 = EQ_MP (SYM th3) th2
     val th5 = EQ_MP (SPEC t NOT_NOT) th4
 in
  th5
 end;
```

Outline

State Transition Systems in HOL

States

- set of states: type *states*
- set of initial states: predicate $\mathscr{B}$
  - $\mathscr{B} : \mathit{states} \to \mathit{bool}$
  - $\mathscr{B}\ s$ means s is an initial state

Transitions

- state transition relation: $\mathscr{R}$
  - $\mathscr{R} : \mathit{states} \times \mathit{states} \to \mathit{bool}$
  - $\mathscr{R}(s, s')$ means s' a successor to s

HOL Definitions

Reachable States

- set of states reachable in at most n steps:
  - $\vdash$ ReachBy 0 $\mathscr{R}$ $\mathscr{B}$ s $=$ $\mathscr{B}$ s
  - $\vdash$ ReachBy $(n+1)$ $\mathscr{R}$ $\mathscr{B}$ s $=$
    ReachBy n $\mathscr{R}$ $\mathscr{B}$ s $\vee$ $\exists u$. ReachBy n $\mathscr{R}$ $\mathscr{B}$ u $\wedge$ $\mathscr{R}(u, s)$
- set of reachable states:
  - $\vdash$ Reachable $\mathscr{R}$ $\mathscr{B}$ s $=$ $\exists n$. ReachBy n $\mathscr{R}$ $\mathscr{B}$ s

Checking Safety Properties

- check $\mathscr{M}$ $\models$ P with the help of
  $\forall s.\ (\exists n.$ ReachBy n $\mathscr{R}$ $\mathscr{B}$ $s) \Rightarrow P\ s$
- use BDDs: compute BDD and check result if the BDD of ‘true‘

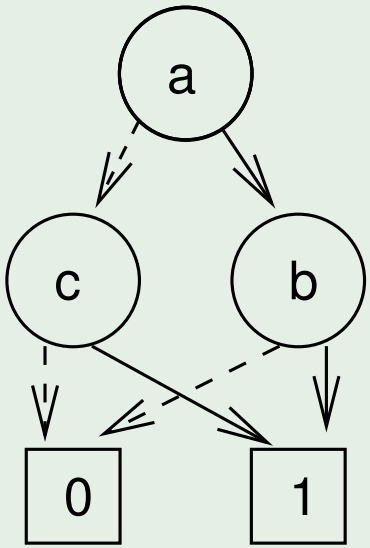
Computing ReachBy $n\ \mathscr{R}\ \mathscr{B}\ s$

Fixpoint Iteration

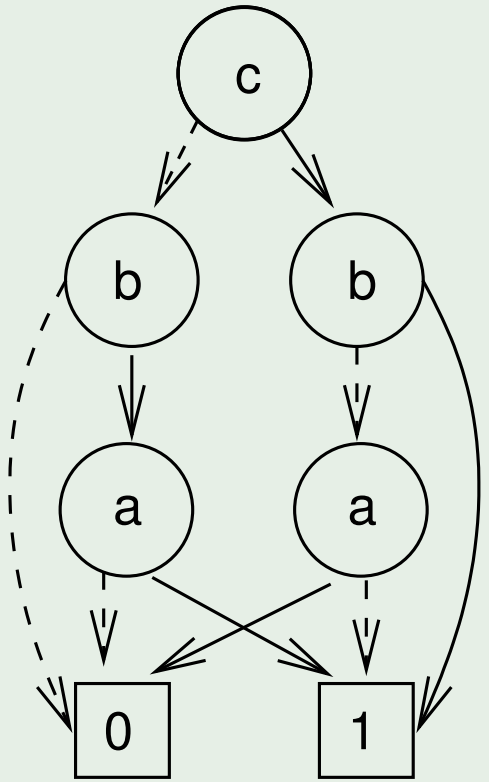
- $\forall s.\ (\exists n.\ \text{ReachBy}\ n\ \mathscr{R}\ \mathscr{B}\ s)\ \Rightarrow\ P\ s$
  is not a quantified boolean formula (QBF)
- Key property:
  $\vdash\ (\text{ReachBy}\ n\ \mathscr{R}\ \mathscr{B}\ s\ =\ \text{ReachBy}\ (n{+}1)\ \mathscr{R}\ \mathscr{B}\ s) \Rightarrow$
  $(\text{Reachable}\ \mathscr{R}\ \mathscr{B}\ s\ =\ \text{ReachBy}\ n\ \mathscr{R}\ \mathscr{B}\ s)$
- Compute Reachable $\mathscr{R}\ \mathscr{B}\ s$ by iteratively computing:
  ReachBy 0 $\mathscr{R}\ \mathscr{B}\ s$
  ReachBy 1 $\mathscr{R}\ \mathscr{B}\ s$
  $\vdots$
  ReachBy $n\ \mathscr{R}\ \mathscr{B}\ s$
  ReachBy $(n{+}1)\ \mathscr{R}\ \mathscr{B}\ s$

Binary Decision Diagrams

Example (BDD for $(a \wedge b) \vee (\neg a \wedge c)$)



variable order $a < b < c$



variable order $c < b < a$

Applying the LCF Approach to BDD Calculation

Consider Judgements (ρ, t, b)

- structure
  - ρ represents a variable order,
  - t is a boolean term all of whose free variables are Boolean
  - b is a BDD.
- such a judgement is valid ($\rho\, t \mapsto b$) if
  - the BDD representing t with respect to ρ
  - is b

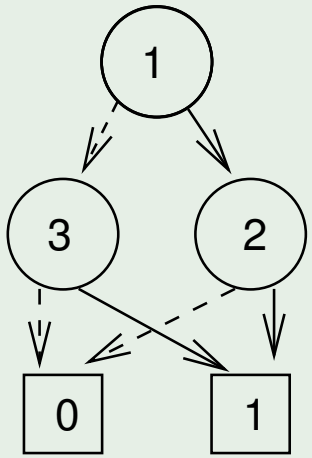
Analogy

- abstract type term_bdd that models judgements
- higher level tools, such as model checkers, are programmed in ML as derived rules

BDD Judgements

Example

$\{a \mapsto 1, b \mapsto 2, c \mapsto 3\} \quad (a \wedge b) \vee (\neg a \wedge c) \mapsto$



(variable order: $a < b < c$)

- can also be written as $BDD(1 \Rightarrow (2 \Rightarrow 1|0)|(3 \Rightarrow 1|0))$

Implementation in HOL `HolBddLib`

BDD Library

- An ML type *termbdd* to represent judgements $\rho\ t \mapsto b$
  - analogous to LCF type *thm* representing logic theorems $\vdash\ t$
- ML functions corresponding to inference rules, for example:

```
BddT            : termbdd
BddNot          : termbdd -> termbdd
BddAnd          : termbdd * termbdd -> termbdd
BddEqualTest    : termbdd -> termbdd -> bool
BddEqMpSYM      : thm -> termbdd -> termbdd
BddThmOracle    : termbdd -> thm
```

Reasoning about BDD Representations of Terms

BDD Derivation

- $\rho\ t \mapsto b$ means term t is represented by BDD b w.r.t. ρ
- let ρ be a map from variables to ordered BDD variable nodes

$$\mathtt{T}\ \frac{}{\rho\ \mathsf{T} \mapsto \mathit{BDD}(1)} \qquad \mathtt{F}\ \frac{}{\rho\ \mathsf{F} \mapsto \mathit{BDD}(0)} \qquad \mathtt{VAR}\ \frac{\rho(v) = n}{\rho\ v \mapsto \mathit{BDD}(n \Rightarrow 1|0)}$$

$$\mathtt{AND}\ \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ t_1 \wedge t_2 \mapsto b_1\ \mathtt{AND}\ b_2} \qquad \mathtt{EQ}\ \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ t_1 = t_2 \mapsto b_1\ \mathtt{EQ}\ b_2}$$

$$\mathtt{IMP}\ \frac{\rho\ t_1 \mapsto b_1 \qquad \rho\ t_2 \mapsto b_2}{\rho\ t_1 \Rightarrow t_2 \mapsto b_1\ \mathtt{IMP}\ b_2} \qquad \mathtt{EXISTS}\ \frac{\rho\ t \mapsto b \qquad \rho\ u \mapsto n}{\rho\ \exists u.\ t \mapsto \mathtt{EXISTS}\ n\ b}$$

$$\mathtt{THM}\ \frac{\rho\ t \mapsto \mathit{BDD}(1)}{\vdash\ t} \qquad \mathtt{SUB}\ \frac{\rho\ t_1 \mapsto b \qquad \vdash\ t_1 = t_2}{\rho\ t_2 \mapsto b}$$

Combining BDD Calculation and Deduction

Calculation and Deduction

- If t_1 a QBF then $\rho\ t_1 \mapsto b$ by BDD evaluation
  - logically use rules `T`, `F`, `VAR`, `AND`, `EQ`, `IMP`, `EXISTS`, ...
  - implement efficiently
- Use theorem proving to prove $\vdash\ t_2$
- Combine using bridging rules `THM` and `SUB`
- `THM` is the only rule that creates theorems

Example Deduction using BDDs

Example

Define $\mathscr{S}_n(s) = \text{ReachBy } n\ \mathscr{R}\ \mathscr{B}\ s$.

$\vdots$	$\vdots$
$\rho\ \mathscr{S}_{20}(s) \mapsto b_{20}$	BDD evaluation
$\rho\ \mathscr{S}_{21}(s) \mapsto b_{21}$	BDD evaluation
$\rho\ (\mathscr{S}_{20}(s) = \mathscr{S}_{21}(s)) \mapsto b_{20}\ \texttt{EQ}\ b_{21}$	EQ
$\vdash\ \mathscr{S}_{20}(s) = \mathscr{S}_{21}(s)$	THM assuming b_{20} EQ b_{21} is
$\vdash\ (\mathscr{S}_{20}(s) = \mathscr{S}_{21}(s)) \Rightarrow (\exists n.\ \mathscr{S}_n(s)) = \mathscr{S}_{20}(s)$	Instance of lemma
$\vdash\ (\exists n.\ \mathscr{S}_n(s)) = \mathscr{S}_{20}(s)$	Modus Ponens and lemma
$\rho\ (\exists n.\ \mathscr{S}_n(s)) \mapsto b_{20}$	SUB
$\rho\ P(s) \mapsto b_P$	BDD evaluation
$\rho\ (\exists n.\ \mathscr{S}_n(s)) \Rightarrow P(s) \mapsto b_{20}\ \texttt{IMP}\ b_P$	IMP
$\vdash\ (\exists n.\ \mathscr{S}_n(s)) \Rightarrow P(s)$	THM assuming b_{20} IMP b_P is

Example: Check AG P holds of model $\mathscr{M}$

Example

$$\mathscr{M} \models \mathtt{AG}\ P$$
$$\downarrow$$
$$\forall \sigma.\ \mathscr{M}\ \sigma \Rightarrow \mathtt{AG}\ P\ \sigma$$
$$\downarrow$$
$$\forall \sigma.\ (\mathscr{B}(\sigma\ 0) \wedge \forall n.\ \mathscr{R}(\sigma\ n,\ \sigma(n{+}1))) \Rightarrow \forall n.\ P(\sigma\ n)$$
$$\downarrow$$
$$\forall s\ n.\ \text{ReachBy}\ n\ \mathscr{R}\ \mathscr{B}\ s \Rightarrow P\ s$$
$$\downarrow$$
$$\forall s.\ (\exists n.\ \text{ReachBy}\ n\ \mathscr{R}\ \mathscr{B}\ s) \Rightarrow P\ s$$
$$\square$$

meaning of $\models$

$\mathscr{M} = \text{Machine}(\mathscr{R}, \mathscr{B})$

definition of ReachBy

first order logic

BDD calculation

Properties

- Reachable $\mathscr{R}\ \mathscr{B}\ s \Rightarrow \mathscr{Q}\ s$ means $\mathscr{Q}$ true in all reachable states
- Might want to verify other properties, e.g:
  - *DeviceEnabled* is always true somewhere along every path starting anywhere (i.e. infinitely often along every path)
  - From any state it is possible to get to a state for which *Restart* holds
  - $\mathscr{Q}$ is true on all paths sometime between i units of time later and j units of time later.
- CTL is a logic for expressing such properties
- Exist efficient algorithms for checking them
- Model checking:
  - check property in a model
  - Emerson & Clarke, early 1980s
  - starting to be used in industry

Computation Tree Logic (CTL)

Syntax of CTL Well-Formed Formulae

wff	$::=$	P	(Atomic formula)
	$\mid$	$\neg wff$	(Negation)
	$\mid$	$wff_1 \wedge wff_2$	(Conjunction)
	$\mid$	$wff_1 \vee wff_2$	(Disjunction)
	$\mid$	$wff_1 \rightarrow wff_2$	(Implication)
	$\mid$	$\mathbf{AX} wff$	(All successors)
	$\mid$	$\mathbf{EX} wff$	(Some successors)
	$\mid$	$\mathbf{A}[wff_1\ \mathbf{U}\ wff_2]$	(Until – along all paths)
	$\mid$	$\mathbf{E}[wff_1\ \mathbf{U}\ wff_2]$	(Until – along some path)

Branching Time Logic

- property Φ hold along all paths: $A\Phi$
- property Φ holds along some paths: $E\Phi$

Paths

Paths

- Let $\mathscr{R}$ have type $\alpha \times \alpha \rightarrow bool$
  - α ranges (intuitively) over states
- An $\mathscr{R}$-path is a function $\sigma : num \rightarrow \alpha$ such that:
  $\forall t.\ \mathscr{R}(\sigma(t), \sigma(t+1))$
- $PATH(\mathscr{R}, s)\sigma$ means σ is an $\mathscr{R}$-path from s
  $PATH(\mathscr{R}, s)\sigma = (\sigma(0)=s) \wedge \forall t.\ \mathscr{R}(\sigma(t), \sigma(t+1))$

Semantic Embedding of CTL in HOL

- Define:

$$
\begin{aligned}
\mathrm{Atom}(p) &= \lambda(\mathscr{R}, s).\ p(s) \\
\neg P &= \lambda(\mathscr{R}, s).\ \neg(P(\mathscr{R}, s)) \\
P \wedge Q &= \lambda(\mathscr{R}, s).\ P(\mathscr{R}, s) \wedge Q(\mathscr{R}, s) \\
P \vee Q &= \lambda(\mathscr{R}, s).\ P(\mathscr{R}, s) \vee Q(\mathscr{R}, s) \\
P \rightarrow Q &= \lambda(\mathscr{R}, s).\ P(\mathscr{R}, s) \Rightarrow Q(\mathscr{R}, s) \\
\mathbf{AX}P &= \lambda(\mathscr{R}, s).\ \forall s'.\ \mathscr{R}(s, s') \Rightarrow P(\mathscr{R}, s') \\
\mathbf{EX}P &= \lambda(\mathscr{R}, s).\ \exists s'.\ \mathscr{R}(s, s') \wedge P(\mathscr{R}, s') \\
\mathbf{A}[P\ \mathbf{U}\ Q] &= \lambda(\mathscr{R}, s).\ \forall \sigma.\ PATH(\mathscr{R}, s)\sigma \\
&\qquad \Rightarrow \\
&\qquad \exists i.\ Q(\mathscr{R}, \sigma(i)) \wedge \forall j.\ j < i \Rightarrow P(\mathscr{R}, \sigma(j)) \\
\mathbf{E}[P\ \mathbf{U}\ Q] &= \lambda(\mathscr{R}, s).\ \exists \sigma.\ PATH(\mathscr{R}, s)\sigma \\
&\qquad \wedge \\
&\qquad \exists i.\ Q(\mathscr{R}, \sigma(i)) \wedge \forall j.\ j < i \Rightarrow P(\mathscr{R}, \sigma(j))
\end{aligned}
$$

AF

Additional Operators

- Example: $\mathbf{AF}P = \mathbf{A}\,[\,\mathsf{T}\ \mathbf{U}\ P\,]$
- $\mathbf{AF}P$ is true if P holds somewhere along every $\mathscr{R}$-path
  - i.e. P is *inevitable*
- Derivation is easy

$$
\begin{aligned}
&\mathbf{AF}P \\
&\quad = \mathbf{A}\,[\,\mathsf{T}\ \mathbf{U}\ P\,] \\
&\quad = \lambda(\mathscr{R}, s).\ \forall \sigma.\ \mathit{PATH}(\mathscr{R}, s)\sigma \ \Rightarrow\ \exists i.\ P(\mathscr{R}, \sigma(i)) \ \wedge\ \forall j.\ j < i \ \Rightarrow\ \mathsf{T}(\mathscr{R}, \sigma(j)) \\
&\quad = \lambda(\mathscr{R}, s).\ \forall \sigma.\ \mathit{PATH}(\mathscr{R}, s)\sigma \ \Rightarrow\ \exists i.\ P(\mathscr{R}, \sigma(i))
\end{aligned}
$$

Example CTL Formulas

EF($Started \wedge \neg Ready$)

It is possible to get to a state where *Started* holds but *Ready* does not hold.

AG($Req \rightarrow$ **AF**Ack)

If a request *Req* occurs, then it will eventually be acknowledged by *Ack*.

AG(**AF***DeviceEnabled*)

DeviceEnabled is always true somewhere along every path starting anywhere: i.e. *DeviceEnabled* holds infinitely often along every path.

AG(**EF***Restart*)

From any state it is possible to get to a state for which *Restart* holds.

Verification and Counterexamples

Typical Safety Question

- is $\mathscr{Q}$ true in all reachable states?
- i.e. is Reachable $\mathscr{R}\ \mathscr{B}\ s \Rightarrow \mathscr{Q}\ s$ true?

Computation

- Compute BDD of Reachable $\mathscr{R}\ \mathscr{B}\ s \Rightarrow \mathscr{Q}\ s$
- if answer *false*: can get counterexample

Generating Counterexample Traces

Finding a Counterexample

- suppose Reachable $\mathscr{R}\ \mathscr{B}\ s \Rightarrow \mathscr{Q}\ s$ is false
- maybe counterexample before fixedpoint
- first find counterexample
  - generate BDDs of ReachBy $i\ \mathscr{R}\ \mathscr{B}$ $(i = 0, 1, \ldots)$
  - at each stage check whether $\mathscr{Q}\ s$ holds
  - hence find smallest n and state s_n such that ReachBy $n\ \mathscr{R}\ \mathscr{B}\ s_n \wedge \neg(Q\ s_n)$
- Then trace backwards using:

  $\text{Pre}\ \mathscr{R}\ \mathscr{Q}\ s = \exists s'.\ \mathscr{R}(s, s') \wedge \mathscr{Q}\ s'$

  $\text{Eq}\ s_1\ s_2 = (s_1 = s_2)$

  - use BDDs to get $s_n, \ldots, s_0$ where ReachBy $(i-1)\ \mathscr{R}\ \mathscr{B}\ s_{i-1} \wedge \text{Pre}\ \mathscr{R}\ (\text{Eq}\ s_i)\ s_{i-1}$
  - Pre $\mathscr{R}\ \mathscr{Q}\ s$ can be deductively simplified (so that BDD of $\mathscr{R}$ not needed)

Model Checking in HOL

Model Checking

- e.g. **AG P**$(\mathscr{R}, \mathbf{s_0})$ is $\forall s.$ Reachable $\mathscr{R}$ (Eq s_0) $s \Rightarrow P\ s$.
- current work has built a CTL model checker inside HOL

HOL + BDD Results

- deduction can enhance state enumeration
- simplify formulas to eliminate subterms
- state enumeration can enhance deduction
- find counterexamples
- formulas for reachable states

Summary

Model Checking Implementation

- deduction using theorem prover written in ML (HOL)
  - extended with `THM` as oracle
- external calls to BDD package written in C (BuDDy)
  - incremental database of computed instances of $\rho\ t \mapsto b$
- model checking by HOL deduction + BDD calculation
  - implemented by deduction rules programmed in ML
  - next 700 formal verification tools ...

Outline

Conclusion

Connect External Tools

- LCF approach and oracles
- tags to trace origins of theorems
- very efficient approach

Implement Tools Internally

- implementation in higher-order logic
- clean and flexible approach