Guardol: A domain-specific language for guards supporting strong automated formal analysis

Andrew Gacek *RockwellCollins*
Konrad Slind *RockwellCollins*
David Hardin *RockwellCollins*
Mike Whalen *U.Minnesota*

May 5, 2011

Part 1

Guards and Guardol

Guard technology

A **guard** mediates information sharing between security domains according to a specified policy.

Typical guard operations on a packet stream:

- read field values in a packet
- change fields in a packet
- transform packet by adding new fields
- drop fields from a packet
- construct audit messages
- remove entire packet from stream

Guard technology at Rockwell-Collins

Rockwell-Collins has some experience in the area (reported at previous HCSS meetings)

- 2005: High assurance guard demo
- 2007: Turnstile guard based on AAMP7
  - AAMP7 based
  - designed, built, delivered
  - undergoing accreditation
- 2010: MicroTurnstile
  - used to guard USB comms in soldier systems
  - AAMP7 based
  - matchbox size

We now seek to make the process of specifying, implementing, and certifying high assurance guards more efficient, flexible, and retargetable.

The Guardol language

Our approach is to develop a **domain-specific language** for guards, plus support technology.

- Automatic generation of implementation and formal analysis artifacts
- Integrate and highly automate formal analysis
- Support a wide variety of guard platforms
- Ability to glue together existing or mandated functionality

The Guardol language

Roughly: **Guardol** $=$ **Ada** $+$ **ML**

Ada provides a familiar setting for our target programmers.

ML types succinctly capture tree-structured data, *e.g.*, email, XML

Guardol language summary

Guardol is a simple conventional imperative language with ML-style datatypes.

- base types (**bool**,**int**,**word32**,**string**)
- record types
- mutual, nested recursive types
- pattern-matching
- second order functions (via externals)
- package system
- specifications integrated into programs

What Guardol doesn't have

- **no** polymorphism
- **no** pointers
- **no** I/O

Example 1: Swap

```
package SwapModule =
begin

function swap (a : in out int, b : in out int) =
 begin
   var tmp : int;
 in
   tmp := a;
   a := b;
   b := tmp;
 end

end
```

This is not a guard

Example 2: Tree Guard

Examines and possibly transforms a tree of messages (strings), calling out to an external *dirty-word-search* function.

```
package DWSTree =
begin
 type Msg = string;

 type Tree =
 { Leaf
 | Node: [Value: Msg;  Left: Tree;  Right: Tree]
 };

 type MsgResult  = {Ok: Msg | Audit: string};

 type TreeResult = {TreeOk: Tree | TreeAudit: string};

 imported function
   DIRTY_WORD_SEARCH(Text:in Msg, Output:out MsgResult);
```

Tree Guard function

```
function Guard (Input : in Tree, Output : out TreeResult) =
begin
var
   ValueResult : MsgResult;
   LeftResult,RightResult  : TreeResult;
in
match Input with
begin
  Tree'Leaf => Output := TreeResult'TreeOk(Tree'Leaf);
  Tree'Node node =>
  begin
    DIRTY_WORD_SEARCH(node.Value, ValueResult);
    match ValueResult with
    begin
      MsgResult'Audit A  => Output := TreeResult'TreeAudit(A);
      MsgResult'Ok ValueMsg =>
      begin
        Guard (node.Left, LeftResult);
        match LeftResult with
        begin
          TreeResult'TreeAudit A => Output := LeftResult;
          TreeResult'Ok LeftTree =>
          begin
            Guard (node.Right, RightResult);
            match RightResult with
            begin
              TreeResult'TreeAudit A => Output := RightResult;
              TreeResult'Ok RightTree =>
                   Output := TreeResult'TreeOk(Tree'Node
                                [ Value: ValueMsg, Left : LeftTree, Right: RightTree ]);
end end end end end end end
end package
```

Control flow via matching

```
...
in
match Input with
begin
   Tree'Leaf => Output := TreeResult'TreeOk(Tree'Leaf);
   Tree'Node node =>
   begin
      DIRTY_WORD_SEARCH(node.Value, ValueResult);
      match ValueResult with
      begin
         MsgResult'Audit A  => Output := ...
         MsgResult'Ok ValueMsg => ...
```

Constructed data

```
...
in
match Input with
begin
   Tree'Leaf => Output := TreeResult'TreeOk(Tree'Leaf);
   Tree'Node node =>
         ....
   TreeResult'Ok RightTree =>
   Output :=
   TreeResult'TreeOk(Tree'Node
      [Value:ValueMsg,Left:LeftTree, Right:RightTree])
...
```

Second order procedures/parameterized modules

```
...
in
match Input with
begin
   Tree'Leaf => Output := TreeResult'TreeOk(Tree'Leaf);
   Tree'Node node =>
   begin
      DIRTY_WORD_SEARCH(node.Value, ValueResult);
      match ValueResult with
      begin
   ...
```

Specification of properties

We add a specification to the Swap module:

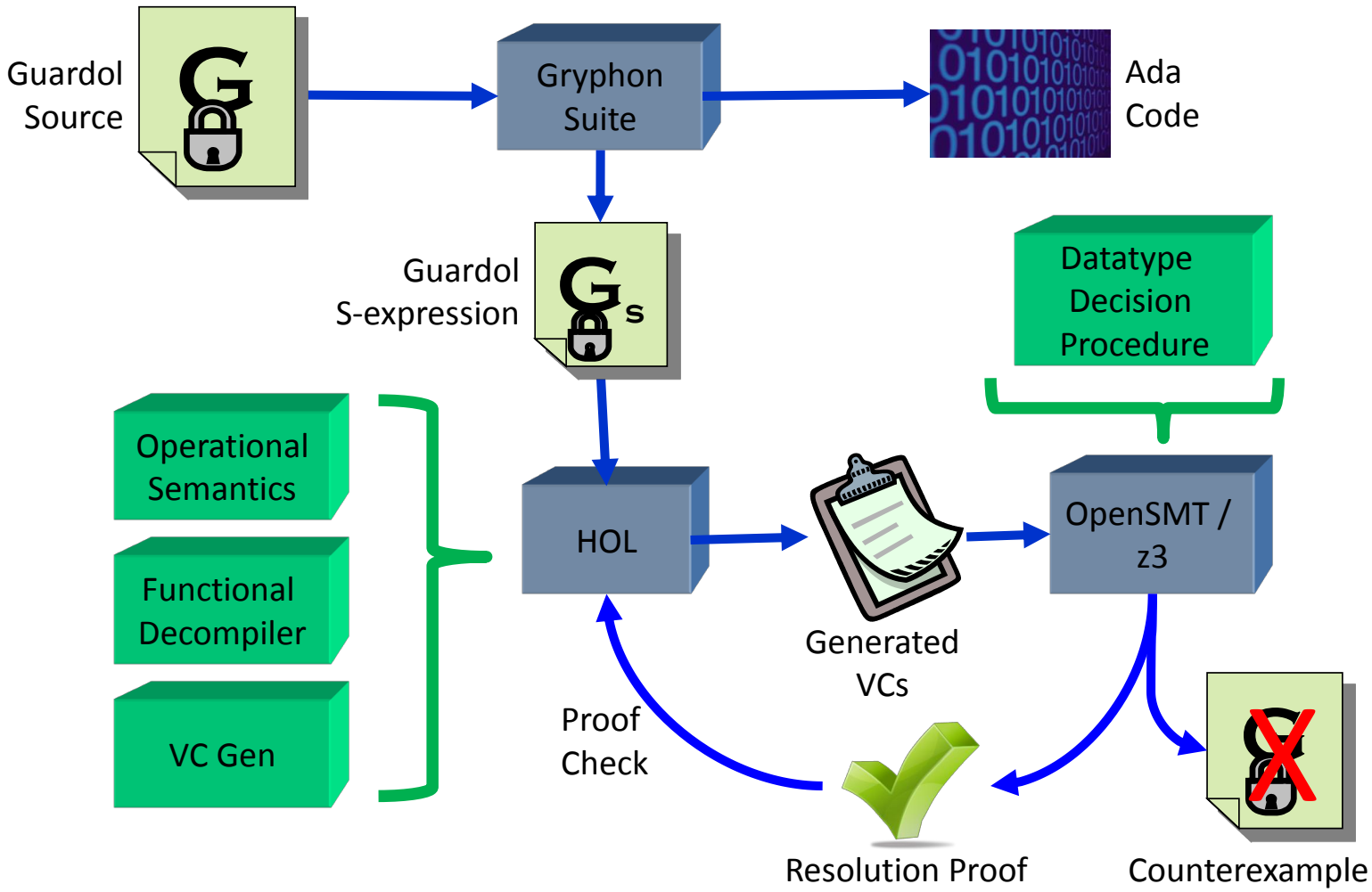
```
package SwapModule = begin

function swap (a : in out int, b : in out int) = ....

spec reswap =
 begin
   var x, y, x0, y0 : int;
 in
  x0 := x; y0 := y;
  swap (x,y);
  swap (x,y);
  check x = x0 and y = y0;
 end
end package
```

Design aspects

- We adopted an intermediate language from **Gryphon** as the basis for Guardol
  - "Software model checking takes off", Miller, Whalen, Cofer, CACM Feb 2010
- Leverages existing code-generation facilities of Gryphon
- Little emphasis on incorporating cutting edge **programming language** features
- Instead, we have a simple language and aim at cutting edge **proof** support

Guardol System Overview

System Diagram



System Components

- Front end
  - Gryphon toolchain
  - parsing, type-checking
  - Ada code generation
- Semantic representation
  - Theory of programs/evaluation
  - Translation to intermediate form
  - Derive functional footprints
- Automatic proof

Code generation

Currently we generate Ada from Guardol programs.

- Extended and adapted existing Gryphon code generator
- Base types directly translate to Ada types
- Recursive datatypes implemented with reference counting (smart pointers)
- Pattern matching translated to Ada case statements (not the same thing!)

Code generation cont'd

- External types and functions result in Ada spec files being generated
- **BUT** implementations for such need to be supplied by the Ada programmer
- IO 'driver' also needs to be implemented by the Ada programmer
- Can also generate rules for Turnstile guard

Semantic representation

In order to reason about Guardol programs we need to have a semantics for programs.

- Guardol semantics formalized in higher order logic
- AST of programs + Operational Semantics
- Translation of programs into equivalent footprint functions
- Translation of functions + specifications to SMT

Automated reasoning about Guardol programs

Our setting provides

- Unbounded user-declared data
- Recursive programs

Although theoretically impossible, we want to automate much or all of the reasoning about Guardol programs.

New decision procedures have recently emerged (Suter-Kuncak) and we have been implementing them.

Logical Basis

Guardol program formally

We formalize the evaluation of Guardol programs in HOL by adapting a quite general semantics due to Norbert Schirmer.

- "Verification of Sequential Imperative Programs in Isabelle/HOL", Norbert Schirmer, TU Munich, 2006.

Datatype of program ASTs:

```
prog
   = Skip
   | Basic of 'a -> 'a
   | Seq of prog * prog
   | Cond of ('a -> bool) * prog * prog
   | Call of 'b
   | withState of ('a -> prog)
   | Throw
   | Catch of prog * prog
   | While of ('a -> bool) * prog
   | Guard of 'c * ('a -> bool) * prog
   | Spec of 'a -> 'a -> bool
```

Guardol programs formally cont'd

Currently we are using the following subset:

```
prog
    = Skip
    | Basic of 'state -> 'state
    | Seq of prog * prog
    | Cond of ('state -> bool) * prog * prog
    | Call of string * string
    | withState of ('state -> prog)
```

Notes:

- Assignment (and other things) represented by **Basic**
- State type is polymorphic: can be any HOL type!
- **Call** takes the name of a procedure to call
- **withState** is exotic and very powerful
- Some are superfluous, *e.g.*, **While**
- Some we don't use yet (but will)

Evaluation

Evaluation is defined using a big-step operational semantics.

Phrased in terms of a *mode* of evaluation. The evaluation state is either in **Normal** mode, or in one of a set of abnormal modes.

$$\textbf{STEPS} : \mathit{env} \to \textbf{program} \to \textbf{mode} \to \textbf{mode}$$

Example:

$$\textbf{STEPS}\ \Gamma\ \textbf{Skip}\ (\textbf{Normal}\ s)\ (\textbf{Normal}\ s)$$

Sequential evaluation

SEQ

$$\frac{\textbf{STEPS}\ \Gamma\ c_1\,(\textbf{Normal}\ s_1)\ s_2 \qquad \textbf{STEPS}\ \Gamma\ c_2\ s_2\ s_3}{\textbf{STEPS}\ \Gamma\ (\textbf{Seq}\ c_1\ c_2)\ (\textbf{Normal}\ s_1)\ s_3}$$

Procedure call

$$\textsc{Call}\ \frac{\Gamma(\mathit{proc}) = c \qquad \mathit{proc} \in \mathbf{Dom}(\Gamma) \qquad \mathbf{STEPS}\ \Gamma\ c\ (\mathbf{Normal}\ s_1)\ s_2}{\mathbf{STEPS}\ \Gamma\ (\mathbf{Call}\ \mathit{proc})\ (\mathbf{Normal}\ s_1)\ s_2}$$

Non-fixed-size commands

Blocks and procedure call are derived notions, using **withState** in a clever way.

$$\textsc{withState}\ \frac{\textbf{STEPS}\ \Gamma\ (f\ s_1)\ (\textbf{Normal}\ s_1)\ s_2}{\textbf{STEPS}\ \Gamma\ (\textbf{withState}\ f)\ (\textbf{Normal}\ s_1)\ s_2}$$

Technical comment (modelling types)

Guardol provides user-definable types.

If we were doing a typical exercise in operational semantics, we would probably formalize the type **system** of Guardol.

But our principal use of this operational semantics is as a basis for reasoning about **individual** programs

So we instead rely on the fact that HOL can define any type in Guardol, *i.e.*, Guardol types are shallowly embedded.

Translation of programs

A Guardol program is translated as follows:

- Guardol base types map to existing HOL types
- Guardol datatypes map to HOL datatype declarations
- All variables are put into a hierarchical state record
- Assignments are state-transforming functions
- Blocks and procedure calls are macro-expanded with pre-and-post assignments to handle argument passing and local variables

Technical comment (modelling state)

The state of a program at any point in execution is the values of all the variables in the program.

Finite maps (or assoc-list) commonly used.

Instead we use records. Variable reads and writes are just record field access and update.

- To access a procedure in a state: **s**.**p**
- To access a variable in a state: **s**.**p**.**v**
- To update a variable: s **with p**.**v** $= x \ldots$

Swap body translation

```
Block (\s1. s1)
 (Seq
  (Basic (\st. st with swap.tmp = st.swap.a))
 (Seq
  (Basic (\st. st with swap.a = st.swap.b))
  (Basic (\st. st with swap.b = st.swap.tmp))))
 (\s1 s2.
  s1 with
      swap.b = s2.swap.b, swap.a = s2.swap.a)
```

From operational semantics to functions

Now we are confronted by a **gaping chasm** when trying to do automated proofs about specific Guardol programs

- An AST having an operational semantics
- SMT proof systems which do not understand operational semantics

The program needs to be **liberated** from the semantics, yet we need the semantics to make sure that any theorems proved pertain to the Guardol program.

Decompilation theorems

- "Formal verification of machine-code programs", Magnus Myreen, Cambridge University, 2008.

A decompilation theorem has the form

$$
\begin{array}{l}
s_1.p.v_1 = v_1 \wedge \ldots \wedge s_1.p.v_k = v_k \wedge \\
\textbf{STEPS}\ \Gamma\ \boxed{\textbf{code}}\ (\textbf{Normal}\ s_1)\ (\textbf{Normal}\ s_2) \\
\Rightarrow \\
\textbf{let}\ (o_1, \ldots, o_n) = \boxed{\textbf{fn}}\ (v_1, \ldots, v_k) \\
\textbf{in} \\
\quad s_2 = s_1\ \textbf{with}\ p.o_1 = o_1, \ldots, p.o_n = o_n
\end{array}
$$

It's just a stylized Hoare triple

The footprint function **fn** is the mathematical function that **code** computes

Decompilation cont'd

Decompilation theorems are automatically proved.

- Bottom-up pass from leaves of AST
- Code and function it computes are built-up simultaneously, using code and functions from child AST nodes

Decompilation cont'd

For example, the decompilation theorem for $c_1; c_2$ is (roughly)

$$
\begin{array}{l}
\mathit{inspec} \wedge \\
\textbf{STEPS}\ \Gamma\ (c_1; c_2)\ (\textbf{Normal}\ s_1)\ (\textbf{Normal}\ s_2) \\
\Rightarrow \\
\textbf{let}\ \mathit{ovars} = (\mathit{fn}_2 \circ \mathit{fn}_1)(\mathit{invars}) \\
\textbf{in} \\
\quad s_2 = s_1\ \textbf{with}\ \mathit{ovars}\ \mathit{updates}
\end{array}
$$

where $\mathit{fn}_1, \mathit{fn}_2$ are the synthesized footprint functions for c_1, c_2.

Decompilation for swap

```
|- ∀u1 u2 a b.
    ((u1.swap.a = a) ∧ (u1.swap.b = b)) ∧
     STEPS Gamma swap_body (Normal u1) (Normal u2)
    ⇒
     let (a',b') = swapFn (a,b)
      in
        u2 = u1.swap with a = a', b = b'
```

where the following function definition has been automatically generated

```
|- swapFn(a,b) = let tmp = a in
                 let a = b in
                 let b = tmp
                 in (a,b)
```

Generating proof goals

Recall our specification for swap:

```
spec reswap =
 begin
   var x, y, x0, y0 : int;
 in
x0 := x; y0 := y; swap (x,y); swap (x,y);
  check x = x0 and y = y0;
 end
```

Generating proof goals

We decompile the code up to the check statement and construct the following goal

```
∀u1 u2 x y x0 y0.
  ((u1.reswap.x = x) ∧ (u1.reswap.y = y) ∧
  (u1.reswap.x0 = x0) ∧ (u1.reswap.y0 = y0)) ∧
  STEPS Gamma code (Normal u1) (Normal u2)
⇒
  (u2.reswap.x = u2.reswap.x0) ∧
  (u2.reswap.y = u2.reswap.y0)
```

Generating proof goals

The decompilation theorem itself looks like

```
∀u1 u2 x y x0 y0.
  ((u1.reswap.x = x) ∧ (u1.reswap.y = y)) ∧
  STEPS Gamma code (Normal u1) (Normal u2)
  ⇒
   let (x'',x0,y'',y0) =
       (λ(x,y).
          let x0 = x in
          let y0 = y in
          let (x',y') = swapFn (x,y) in
          let (x'',y'') = swapFn (x',y')
          in (x'',x0,y'',y0)) (x,y)
   in
   u2 = u1.reswap with x = x'', x0 = x0,
                       y = y'', y0 = y0
```

Starting the proof

Now the antecedents of the goal satisfy the antecendents of the decomp. theorem so we can now use **swapFn** to prove the goal.

```
   let (x'',x0,y'',y0) =
      (λ(x,y).
         let x0 = x in
         let y0 = y in
         let (x',y') = swapFn (x,y) in
         let (x'',y'') = swapFn (x',y')
         in (x'',x0,y'',y0)) (x,y)
   in
   u2 = u1.reswap with x = x'', x0 = x0,
                       y = y'', y0 = y0
⇒
  (u2.reswap.x = u2.reswap.x0) ∧
  (u2.reswap.y = u2.reswap.y0)
```

Mapping to SMT

Given a guard, plus a property to prove, we somehow have to reduce it inside HOL until it becomes acceptable to an SMT system.

There are two parts to this:

- Inducting using the induction scheme for the guard. This generates the sub-cases of the goal that don't need induction.
- Reducing the resulting problems to be acceptable to the SMT solver

The first task is a well-known theorem proving activity.

The second task I will skip for lack of time.

Part 4

SMT for recursive programs

Proofs of recursive programs

Systems like ACL2 have been used for eons now to reason about recursive programs over tree-structured data

This is valuable technology for our group and will likely remain so

However, new decision procedures are emerging for functional programs over recursive data

They may provide higher levels of automation

Suter-Kuncak formulas

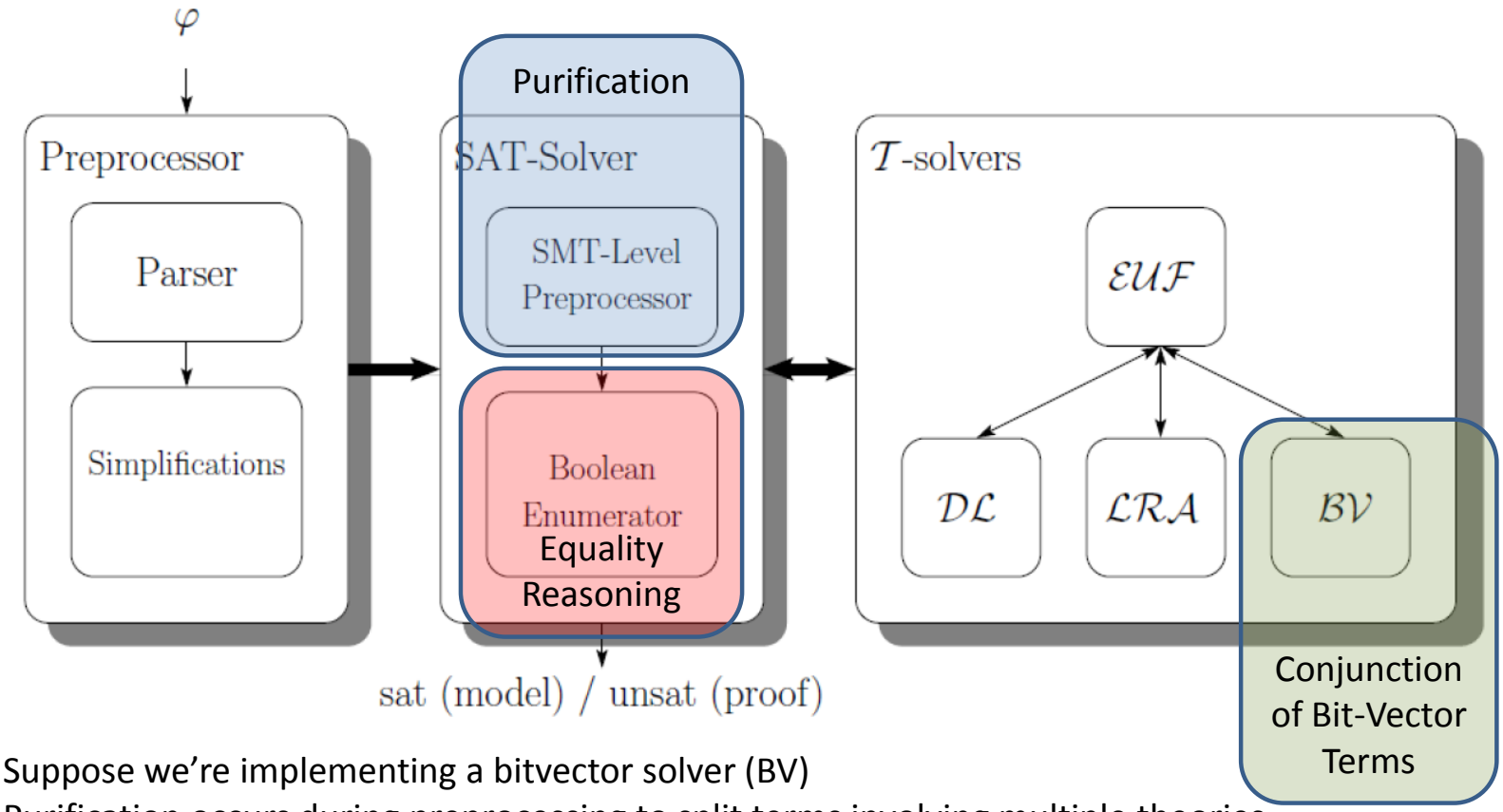
Properties over recursive datatypes, where **catamorphisms** are used to map the data objects into decidable theories.

$$\textbf{cat nil} = e$$
$$\textbf{cat}(h :: t) = \{h\} \cup \textbf{cat}\ t$$

A slightly weaker form of primitive recursion

Contemporary SMT design

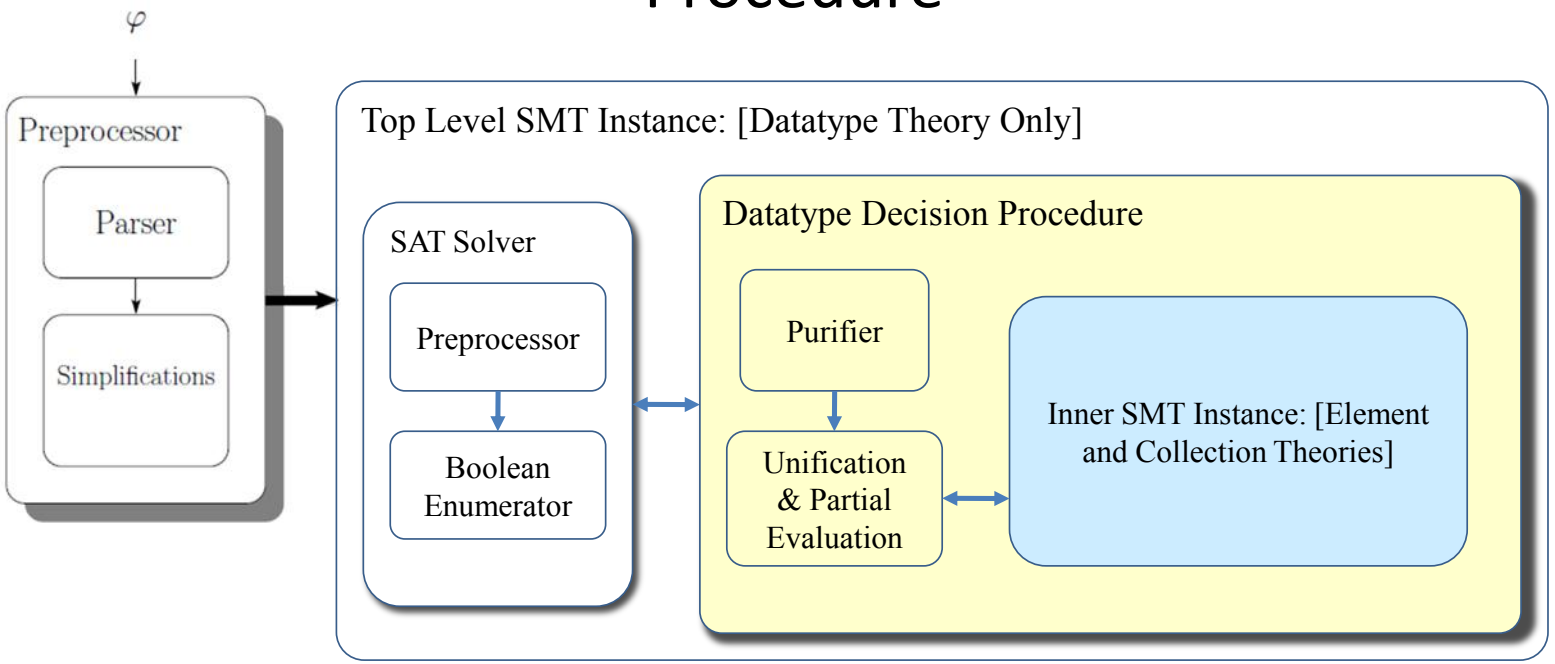
Implementing "Normal" Decision Procedures



Suppose we're implementing a bitvector solver (BV)
Purification occurs during preprocessing to split terms involving multiple theories
BV solver is fed a conjunction of terms ONLY from bitvector theory: (x ^ y ^ z ^ ...)
SAT solver handles equalities "crossing" theories due to purification

Suter-Kuncak design

Implementing Suter & Kuncak Decision Procedure



To create a tool to implement Suter's & Kuncak's procedure we need to modify the standard SMT architecture

The Suter decision procedure is *supervisory*. It operates over terms containing atoms from other theories. Therefore, we forward all theory atoms to it at the top level.

The theory generates SMT-style theories with disjunctions involving terms from other theories; we call an inner instance of OpenSMT to solve the "normal" SMT problem

S-K implementation issues

- No implementations!
- Incomplete approaches, via quantifier instantiation
- Extension to mutual recursion.
- Completeness and its importance.
- Counterexamples

Summary

The Guardol system is a domain-specific language aimed at advancing the state of the art in developing and proving correctness of high-assurance guards.

Guardol programs have their functional equivalents derived by proof

Functional specifications of Guardol programs are soundly and automatically translated into SMT goals.

Status

- Ada code generation works
- Small examples + specs decompiling
- Working on recursive procedures
- Automatic generation of goals for Z3, using quantifiers
- Working on OpenSMT S-K implementation

Future Work

- End-to-end system
- Larger examples
- Decompilation of higher-order packages, type parameterization
- Proof-producing SMT
- Improve specification language
- Improve concrete syntax

THE END